Clube Delphi

Ano 13 :: Edição 152 :: R$ 14,90

DevMedia

Arquitetando um envio de e-mails
Três opções de componentes, uma única solução

Aplicações de banco de dados, quantas camadas possui a sua?
Um estudo sobre 1, 2 e 3 camadas

O NOVO DELPHI XE4

Mobilidade em suas mãos

Serialize informações dinamicamente
Explore objetos pela RTTI

Criando objetos de forma eficiente
Aplicação dos padrões criacionais

ISSN 1517990-7

VOCÊ É ASSINANTE MVP?

ENTÃO PODE TER CERTEZA: O MERCADO PREFERE VOCÊ!

DESENVOLVEDOR ATUALIZADO É O MAIS VALIOSO DO MERCADO!

Muitos profissionais estão acomodados, não se atualizam, não leem livros e não vão a eventos. Acham que é a empresa que deve mantê-los atualizados. Gastam dinheiro com tudo, mas acham caro investir menos de R$60,00 por mês em suas carreiras.

Mas você não! Você é MVP! Parabéns!

TENHA ACESSO A:

- +DE 260 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 7.850 VÍDEO-AULAS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

POR APENAS 59,90 MENSAIS

DevMedia

Acesse: www.devmedia.com.br/mvp

Sumário

Conteúdo sobre Novidades

06 – Delphi XE 4: Uma visão geral das novidades
[*Fabrício Hissao Kawata*]

Artigo do tipo Mentoring

14 – Aplicando boas práticas para criação de objetos
[*Guinther Pauli*]

25 – Aplicando a RTTI para Serializar Informações
[*Diego Garcia*]

Conteúdo sobre Boas Práticas

35 – Envio de e-mails com boas práticas
[*Daniel Laporte e Paulo Quicoli*]

Conteúdo sobre Boas Práticas

43 – Aplicações de banco de dados - 1-tier, 2-tier e N-tier
[*Fabrício Hissao Kawata*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Clube Delphi

Ano 13 - 152ª Edição 2013 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@gmail.com

Equipe Editorial

Guinther Pauli
guintherpauli@gmail.com

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Giuliano Scombatti Pinto
giuliano@sygnux.com.br

Daniel Sobrinho Laporte
daniel.laporte@gmail.com

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Wesley Yamazack (yamazack@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

EDITORIAL

A Embarcadero tem abordado diferentes mercados nos últimos anos. Seu foco recentemente está voltado para o desenvolvimento Mobile. Contudo, um framework que sempre está presente é o DataSnap. Sempre é apresentado como a solução para todos os problemas. Nosso colega Roberto Scheneiders apresenta em seu artigo uma pesquisa comparativa entre o DataSnap, um framework Java, o WCF do .NET Framework e um framework independente desenvolvimento em Delphi chamado mORMot. Todos são colocados à prova como servidores de aplicação, simulando um acesso realmente grande de acessos, simultâneos ou não. O resultado para muitos será uma surpresa, leia o artigo.

O Fabrício continua apresentando para nós as novidades criadas, desta vez temos um overview completo do FireDAC. Veja como utilizá-lo para situações comuns como conexão com Firebird, ordenação de dados em memória, filtros, campos calculados, mestre-detalhe e muito mais. Após esse artigo você estará pronto para começar a utilizar o FireDAC em seus próprios projetos.

Voltando agora os olhos para situações reais de vários desenvolvedores, o Giuliano preparou o que posso chamar "must read" para aqueles que precisam integrar geração e administração de boletos em seus sistemas comerciais. Um braço da suíte ACBr é apresentado, o ACBr Boleto. Além de ser apresentada as funcionalidades, algumas características desse tipo de cobrança são expostas, tirando assim várias dúvidas dos menos experientes no assunto. Não perca.

Outro assunto que tem chamado a atenção dos desenvolvedores Delphi é a questão do ORM, principalmente no uso do framework Aurelius. Quando se fala de ORM é preciso entender muito bem sobre como mapear uma classe, ou um conjunto de classes, para um banco de dados, pois, é nisso que um framework ORM se baseia para gerar o acesso ao banco de dados. Assim, um mapeamento incorreto trará problemas para a aplicação, seja informações incorretas ou até mesmo perda de desempenho. O artigo trazido pelo Murilo faz uma abordagem de vários cenários de mapeamento, clarificando o assunto para quem deseja caminhar essa estrada.

E por fim, temos um artigo de banco de dados, apresentado pelo Matheus Lima. Veja como utilizar recursos específicos do Oracle para criar consultas textuais inteligentes. Entenda o mecanismo de sugestão de textos, similar ao que o Google aplica em consultas, e como utilizar tudo isso com Delphi, em um exemplo simples mas funcional.

Boa leitura!

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

DevMedia

Seu **conhecimento** tem **visibilidade** na DevMedia

Publique artigos e vídeos no site e ganhe mais visibilidade em seu trabalho. Somos o maior site do Brasil para desenvolvedores.

Para publicar é bem simples: mande um email para nosso editor **Wesley Yamazack (yamazack@gmail.com)**.

Para mais informações acesse:
http://www.devmedia.com.br/seja-um-autor-devmedia/28103

DevMedia

Delphi XE4: Uma visão geral das novidades

Confira o que há de mais novo na mais recente versão do Delphi

Alguns anos atrás, ainda na época do lançamento da versão XE2, um dos novos recursos apresentados pelo Delphi foi sua nova plataforma de aplicações, o FireMonkey. Como principal atrativo estava seu caráter multiplataforma, que provia a construção de aplicativos tanto para Windows (32 e 64 bits), quanto para Mac OS X e iOS (**BOX 1**). Para a época, se tratou de uma grande revolução no contexto Delphi, uma vez que a tradicional VCL (**BOX 2**) ganhava uma companhia à altura, ao mesmo tempo em que se expandiam as plataformas suportadas pela ferramenta. Este cenário perdurou até a versão seguinte (XE3), em que o suporte a iOS foi temporariamente cessado. Agora com o lançamento do XE4, não só este suporte é reativado, como também a ferramenta e a própria linguagem ganham novas adições em vista de um total suporte a este tipo de desenvolvimento.

BOX 1. iOS

iOS é o nome do Sistema Operacional móvel da Apple, anteriormente conhecido como iPhone OS. Apesar do indicativo, além do próprio iPhone, atualmente ele está presente também em dispositivos iPad, iPod Touch e Apple TV. Dentre suas características mais peculiares está o seu conceito de interação com o usuário baseada em toques e gestos e sua execução restrita a hardwares de sua mesma proprietária.

BOX 2. VCL

Visual Component Libray, ou simplesmente VCL, é um conjunto (biblioteca) de componentes visuais providos pelo Delphi e voltados à construção de aplicações Windows. A composição de sua arquitetura interna se dá por um emaranhado de classes, as quais originam diversos elementos visuais e não visuais úteis no processo de desenvolvimento.

O atual objetivo da Embarcadero é possibilitar aos desenvolvedores Delphi a criação de aplicativos móveis através da utilização de suas habilidades e conhecimentos já anteriormente adquiridos com o desenvolvimento tradicional Desktop. Por conseguinte, a partir de uma mesma base de código o desenvolvedor será capaz de produzir aplicativos para dispositivos e plataformas distintas. No caso do iOS, pelo fato do Delphi produzir aplicações essencialmente nativas, estas acabam por levar vantagem em relação às aplicações elaboradas a partir de linguagens Web ou baseadas em Script, principalmente por questões de segurança e performance.

Resumo DevMan

Delphi XE4: Uma visão geral das novidades:

O Delphi XE4 é apresentado pela Embarcadero como sendo a ferramenta ideal para o desenvolvimento de aplicações nativas para Windows, Mac e iOS. Isto porque, a partir de uma mesma base de código, que envolve a codificação que todo desenvolvedor Delphi já está acostumado, agora é possível se produzir aplicativos para estas três plataformas. Adicionalmente, este suporte a diferentes plataformas habilita também o desenvolvimento para múltiplos dispositivos, vide PCs (Windows e Mac OS X), Tablets (iOS) e Smartphones (iOS). Sendo assim, o que se pode notar é que neste Release, a grande novidade fica então por conta da volta do suporte a iOS (provido ainda no XE2 e retirado no XE3) e, por conseguinte, a produção de aplicativos para as plataformas relacionadas a ele - Tablets e Smartphones. Obviamente, as novidades desta nova versão do Delphi não se restringem somente ao suporte a iOS, contudo, conforme será visto no decorrer artigo, as principais giram em torno deste contexto e da plataforma de aplicações FireMonkey.

Em que situação o tema é útil:

O tema é útil para demonstrar as recentes novidades e investidas da Embarcadero para com o Delphi XE4. Com as atuais necessidades de mercado, o Delphi XE4 vem fortemente focado nos cenários que mais crescem (desenvolvimento para Smartphones e Tablets). Sendo assim, o tema faz-se necessário para quem deseja estar atualizado com tudo que há de mais moderno no IDE.

FireMonkey FM3

Surgido no XE2, o FireMonkey é a mais recente plataforma de aplicações do Delphi. Diferente da VCL, que se restringe à produção de aplicativos para Windows, o FireMonkey apresenta uma característica cross-plaftorm (multiplataforma). Em razão disso, o Delphi XE4 traz como uma de suas grandes novidades a nova versão do FireMonkey, comercialmente nomeada como FireMonkey FM3, e que é a base para o novo conceito adotado pela ferramenta, que envolve o desenvolvimento multidispositivo e multiplataforma, a partir de uma mesma base de código. Nesta sua nova versão, o FireMonkey apresenta uma série de novidades (recursos, melhorias, classes, componentes e units) relacionadas principalmente ao novo suporte a iOS.

Novas units

Em razão da incorporação de novos recursos, o FireMonkey apresenta uma série de novas units em seu contexto, tais como:

- FMX.WebBrowser;
- FMX.PhoneDialer;
- FMX.Controls.3D;
- FMX.MagnifierGlass;
- FMX.InertialMovement.

Novos componentes

Em complemento, o FireMonkey FM3 apresenta ainda uma gama de novos componentes, muitos deles relacionados (não exclusivos) à plataforma iOS, tais como:

TCameraComponent

Componente não visual que, conforme seu próprio nome sugere, provê acesso aos recursos de câmera e flash de um dispositivo físico. No caso do iOS, a opção pela escolha da câmera frontal ou traseira do aparelho se dá por meio de sua propriedade Kind.

TMagnifierGlass

Componente de lupa para textos e imagens com recursos de escalas e formas (circular e retangular).

TWebBroowser

Componente visual com a mesma proposta do componente TWebBrowser, já tradicional da VCL. Funciona como um navegador de internet, provendo as funções básicas e essenciais de um Web Browser, tais como "navegar", "voltar" e "avançar". Dessa forma, possibilita o carregamento e exibição de conteúdo Web ou de arquivos locais. O endereço da página ou o caminho do arquivo a ser carregado é definido através da propriedade URL do componente.

TListView

Já conhecido da VCL, o TListView é um componente visual destinado à exibição de uma coleção de itens, tal como uma lista. Seu diferencial fica por conta de que cada item da lista pode dispor de uma representação gráfica particular, fazendo uso, por exemplo, de uma imagem associada, um ícone indicativo, um texto complementar, ou mesmo uma simples legenda (Caption). Agora no XE4 é então introduzida uma versão do TListView para FireMonkey, o que imediatamente o torna um dos principais controles para "lista" nestes tipos de projeto.

TListBox

Da mesma forma que o TListView, outro componente visual provindo da VCL que ganha sua versão para FireMonkey é o TListBox. Sua utilização se assemelha a do TListView, mas seu foco se dá sobre listagens simples e curtas.

TLayer3D e TViewport3D

No XE4, o FireMonkey apresenta ainda alguns elementos específicos para trabalhos 3D. Como exemplo, o novo componente TViewport3D é responsável por prover a visualização de objetos 3D. Já o também novo TLayer3D, quando utilizado em um 3D FireMonkey Form, possibilita a colocação de controles gráficos 2D em sua superfície.

TLocationSensor, TMotionSensor e TOrientationSensor

Outra novidade trazida pelo FireMonkey em termos de componentes são os relacionados a sensores, num total de três novos elementos. Eles são disponibilizados na Tool Palette do IDE, num grupo especial denominado Sensors, conforme mostra a **Figura 1**.

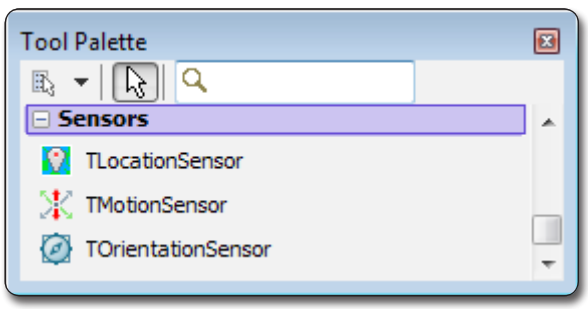

Figura 1. Novos componentes FireMonkey relacionados a sensores

TLocationSensor é um componente especialmente projetado para aplicações FireMonkey Mobile (iOS) e está relacionado à questões como localização e posicionamento geográfico (Ex: GPS). TMotionSensor é um sensor de movimento através do qual se obtém acesso e informações de recursos como acelerômetro, ângulo e estado, da plataforma destino (iOS e Windows). Por fim, TOrientationSensor é o componente que atua como uma bússola ou giroscópio, trabalhando com informações relacionadas a rumo, inclinação e distância.

FireMonkey Mobile Application

FireMonkey Mobile Application é o nome do novo tipo de projeto disponível no IDE do Delphi XE4, destinado à construção de aplicativos para dispositivos móveis, mais especificamente iPhone

e iPad. Dessa forma, se somado aos tipos existentes, agora são três as opções de projetos FireMonkey, cada qual voltada a uma área de interesse diferente:

• FireMonkey Desktop Application: construção de aplicativos para Windows (32 e 64-bits) e Mac OS X;
• FireMonkey Metropolis UI Application: construção de aplicativos no estilo Metro do Windows 8;
• FireMonkey Mobile Application: construção de aplicativos para iOS;

FireMonkey Mobile Wizard

FireMonkey Mobile Wizard é o nome de outra novidade apresentada no Delphi XE4 e que está diretamente relacionada ao novo tipo de projeto FireMonkey Mobile Application. Isto porque, conforme seu próprio nome sugere, o FireMonkey Mobile Wizard nada mais é do que um assistente inteligente, que é imediatamente aberto toda vez que se aciona o tipo de projeto FireMonkey Mobile Application ou a nova opção de formulário FireMonkey Mobile Form. Para o primeiro, o assistente tem uma única função, que é a de prover ao desenvolvedor uma lista de opções de tipos (modelos) de aplicações disponíveis no contexto Mobile, e que servirá como ponto de partida para a criação do aplicativo a ser desenvolvido, conforme mostra a **Figura 2**. Já para o segundo, permite que se escolha um, dentre os dois tipos de formulários disponíveis para aplicações FireMonkey Mobile, conforme mostra a **Figura 3**.

FireMonkey Mobile Application Template

Cada opção disponível no Wizard é então considerada um template ou, mais precisamente, um FireMonkey Mobile Application Template. Em outras palavras, um template caracteriza um modelo pré-definido de aplicação, contendo suas próprias otimizações, bem como seu próprio conjunto de elementos (Forms e componentes) já pré-configurados para seu propósito. Dentre as opções disponíveis, podemos classificá-las em três grupos, de acordo com o tipo de dispositivo que visa atender:

Tablets e Smartphones

• Blank Application: modelo de aplicação vazio;
• 3D Application: modelo vazio para uma aplicação FireMonkey 3D;
• Header/Footer: modelo de aplicação simples, cujo layout é baseado na perspectiva Cabeçalho/Corpo/Rodapé;
• Header/Footer with Navigation: modelo de aplicação semelhante ao anterior, mas cujo layout apresenta um TabControl contendo duas guias (Tabs) para navegação;
• Tabbed: modelo de aplicação cujo layout é composto de guias (Tabs). Por padrão, são criadas inicialmente quatro guias;
• Tabbed with Navigation: neste modelo, o layout pré-definido da aplicação é composto por quatro guias, sendo que a primeira apresenta um controle de guias (TabControl) aninhado;

Somente Smartphones

• Phone Master-Detail: modelo de aplicação voltado para Smartphones. Apresenta um TabControl contendo duas guias (Tabs), sendo uma destinada à listagem principal de dados e outra para as informações de "detalhe" destes;

Somente Tablets

• Tablet Master-Detail: modelo de aplicação voltado para Tablets, cujo layout é dividido em duas partes – listagem principal de dados e informações de detalhe.

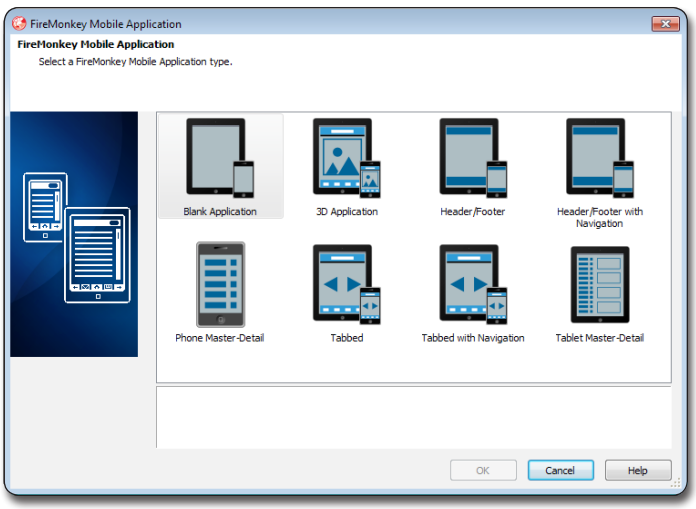

Figura 2. Tipos de aplicações FireMonkey Mobile disponíveis

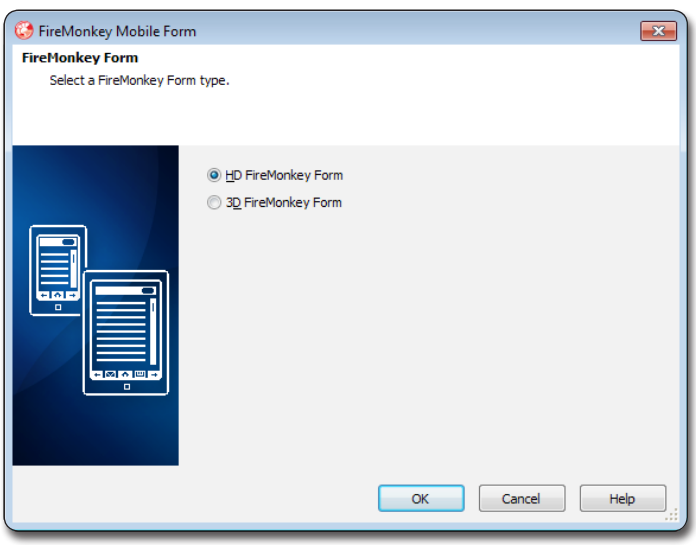

Figura 3. Tipos de formulários FireMonkey Mobile disponíveis

FireMonkey Mobile Form

Conforme já mencionado anteriormente, outra novidade relacionada às aplicações FireMonkey Mobile é o novo formulário, exclusivo para essas aplicações, denominado FireMonkey Mobile Form. Nativamente, ele apresenta dois tipos: HD FireMonkey Form e 3D FireMonkey Form. O primeiro é o mais simples, sendo uma instância da classe TForm do FireMonkey (FMX.Forms), enquanto que o segundo é uma instância da classe TForm3D (FMX.Forms3D).

FireMonkey Mobile Form Designer

Em vista das aplicações FireMonkey Mobile, o Delphi XE4 traz ainda outro novo recurso associado, que é um novo Form Designer exclusivo para o desenvolvimento de aplicativos móveis. Nominalmente, ele é apresentado como FireMonkey Mobile Form Designer, mas nas diversas literaturas espalhadas pela rede também pode ser tratado simplesmente como Mobile Designer. Conceitualmente falando, ele nada mais é do que um Form Designer especial, que é exibido pelo IDE no modo Design de um FireMonkey Mobile Form, conforme mostra a **Figura 4**.

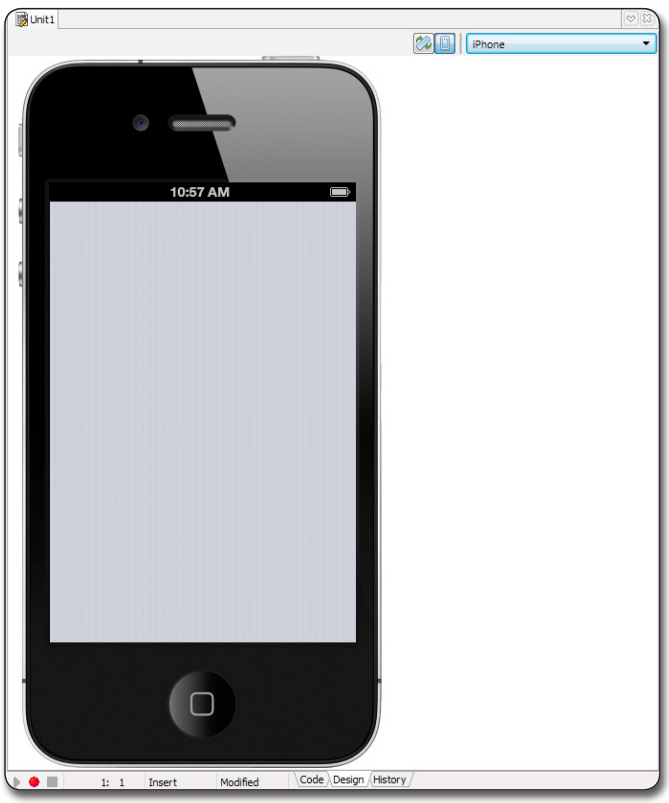

Figura 4. FireMonkey Mobile Form Designer

Ainda no próprio Designer há um combo de seleção na parte superior, que apresenta uma lista de opções ao desenvolvedor. Tal conjunto é então tratado como Design Devices, e que nada mais são do que formas pré-definidas que irão dar uma ideia sobre a aparência final do formulário em determinado tipo de dispositivo. Um recurso similar a este já podia ser encontrado anteriormente em outro IDE da Embarcadero, o RadPHP (atual HTML5 Builder). Num exemplo simples, ao selecionar a opção iPad, o Form mostrado no Designer, apresenta uma dimensão maior, simulando o uso em um dispositivo deste tipo. Toda esta mudança visa contribuir somente no desenvolvimento. Isto porque, o redimensionamento efetivo do Form para o dispositivo (iPhone ou iPad), já ocorre de forma automática ao se fazer o Deploy da aplicação para o aparelho físico, independente da opção de Design Devices utilizada.

FireMonkey Frame

Já muito tradicional na VCL, o Frame é uma espécie de recipiente (container), tal como um Form, que serve para se agrupar componentes, definindo layouts e funcionalidades. Uma particularidade sua é que uma vez construído, ele pode então ser reutilizado dentro de outros Forms do projeto. Sendo assim, são úteis na construção de blocos contendo grupos de controles que são utilizados em diversos pontos da aplicação. Agora na versão XE4 do Delphi, o FireMonkey apresenta suporte a Frames, numa forma de trabalho idêntica ao da VCL. No IDE, agora são duas as opções de Frame, conforme mostra a **Figura 5**.

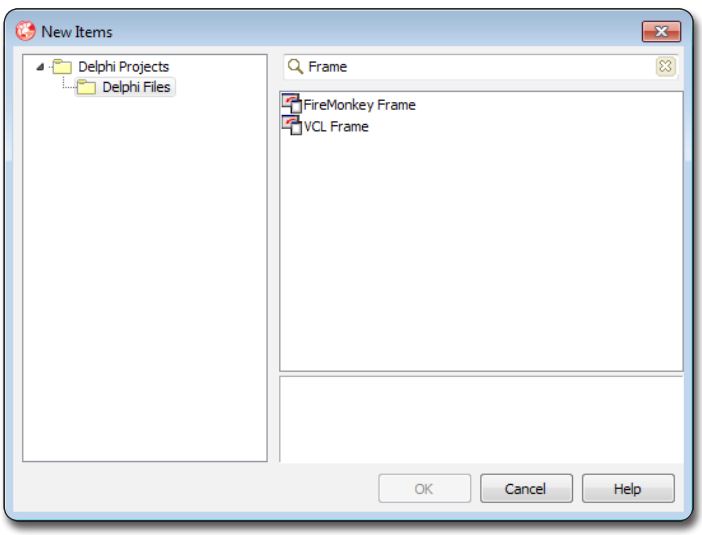

Figura 5. FireMonkey Frame

iOS Mobile Help Wizard

O desenvolvimento iOS no XE4 apresenta algumas particularidades, cuja principal é a necessidade de um ambiente Mac disponível para se realizar os testes da aplicação que está sendo construída. Em suma, apesar desta versão XE4 apresentar os novos compiladores Mobile que possibilitam a compilação de um projeto FireMonkey Mobile Application já no próprio IDE do Delphi, a "execução" e testes do aplicativo só se dão em um simulador/dispositivo físico devidamente instalado/conectado a um Mac. Para ilustrar este cenário, a **Figura 6** mostra um esboço do processo por completo.

Com o intuito de conscientizar o desenvolvedor e assim facilitar toda esta operação, o Delphi XE4 provê um assistente (Wizard), denominado iOS Mobile Help Wizard (**Figura 7**), que é acionado logo durante a criação de um projeto iOS (FireMonkey Mobile Application) no IDE. Seu objetivo é o de ajudar na preparação do ambiente de desenvolvimento para a construção de aplicações móveis para iOS, o que inclui a configuração de um dispositivo Mac com o próprio Delphi.

Target Platforms - iOS

Ainda na versão XE2, o Delphi trouxe como novidade o suporte a múltiplas plataformas, dando um passo adiante ao seu tradi-

cional suporte a Windows 32-bits. Esta fato marcou então a inclusão do já esperado suporte a Windows 64-bits, recurso este que era um desejo antigo da comunidade, além de incorporar ao contexto do Delphi o suporte às plataformas da Apple, vide Mac OS X e iOS. Em vista disso, a estrutura hierárquica de um projeto no Project Manager do IDE ganhou um novo elemento, denominado Target Platforms. Este novo nó fundamentalmente especifica a(s) plataforma(s) de destino do projeto. Obviamente, essa escolha irá variar de acordo com o tipo do projeto corrente. Para um projeto do tipo VCL Forms Application, por exemplo, as opções de plataformas disponíveis se restringem a Win32 e Win64, enquanto que uma aplicação FireMonkey Desktop Application, além destas duas opções, há a possibilidade de escolha pela plataforma Mac OS X.

Agora no Delphi XE4, com a volta do suporte a iOS (extinto na versão XE3), o nó Target Platforms ganha duas novas opções, ambas relacionadas a esta platafoma:

- iOS Device: a plataforma de destino, neste caso, é um aparelho físico rodando iOS. Esta opção provê outra três sub-opções de Build –> Debug (voltado ao ciclo de desenvolvimento e testes reais), Ad-hoc (voltado à distribuição privada) e App Store (para produção da versão final da aplicação a ser enviada à App Store);
- iOS Simulator (**BOX 3**): a plataforma de destino, neste caso, é um software simulador iOS. Novamente há três sub-opções –> iPad (para simulação baseada num aparelho deste tipo), iPhone (para simulação baseada em um aparelho iPhone "genérico") e iPhone 5 (para simulação baseada num aparelho especificamente deste modelo);

Essas opções ficam disponíveis para o também novo tipo de projeto FireMonkey Mobile Application, conforme mostra a **Figura 8**.

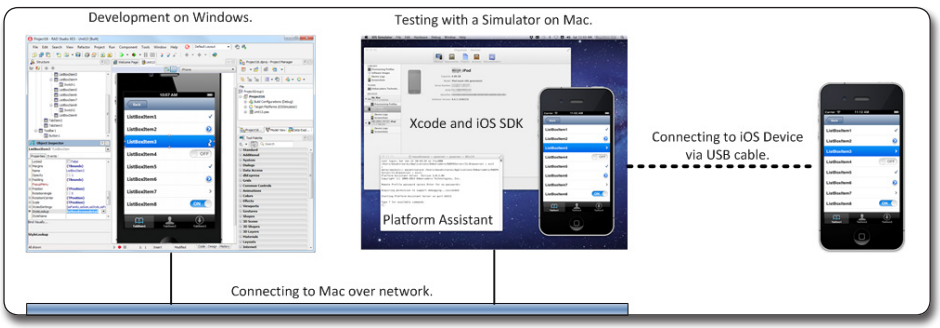

Figura 6. Processo de desenvolvimento para iOS no Delphi (Fonte: Embarcadero)

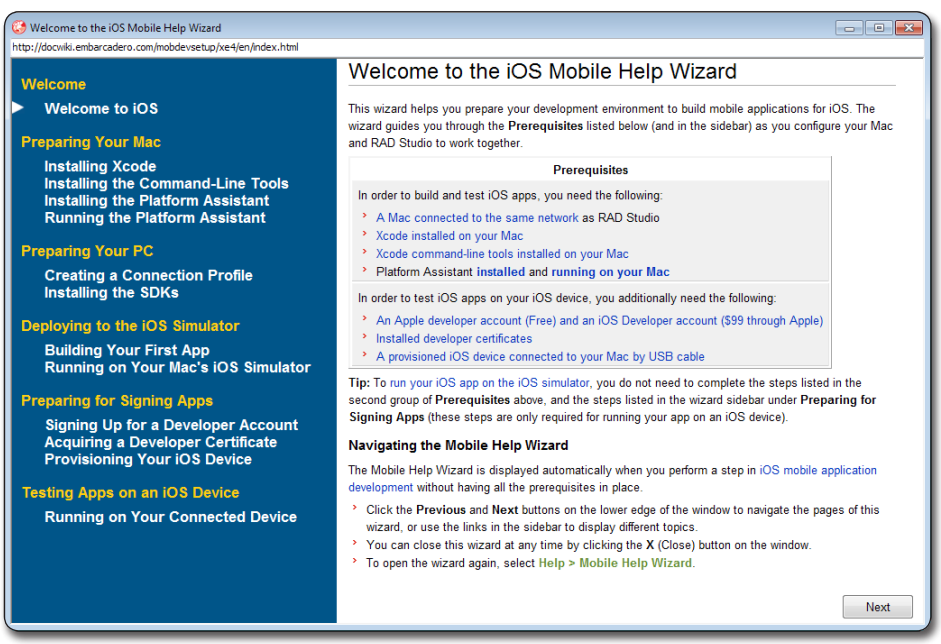

Figura 7. iOS Mobile Help Wizard

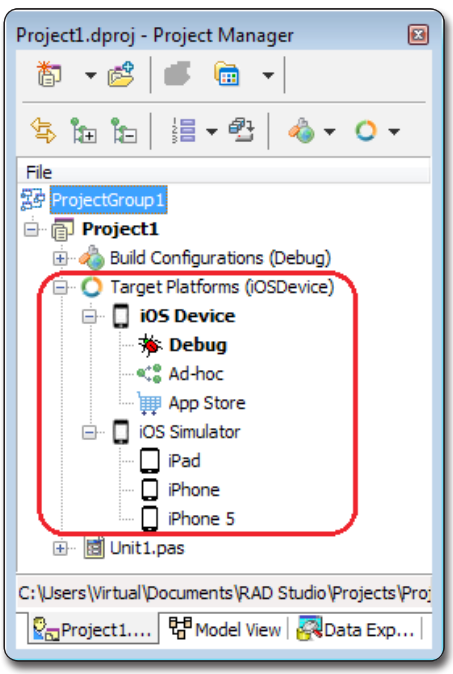

Figura 8. Target Platforms – iOS

BOX 3. iOS Simulator

iOS Simulator é uma plataforma disponibilizada pela Apple, parte integrante das ferramentas XCode que acompanham o SDK do iOS. Seu uso provê ao desenvolvedor, de forma ágil e simplificada, a realização de testes de seus aplicativos construídos para iOS. No processo de desenvolvimento, o iOS Simulator substitui a presença de um dispositivo físico com iOS, podendo simular de forma eficaz vários dispositivos iOS (Ex: iPhone e iPad), bem como várias versões do próprio Sistema Operacional. Sua execução se dá essencialmente sobre um dispositivo Mac.

Novos compiladores – Delphi Mobile Compilers

O Delphi XE4 traz consigo dois novos compiladores, voltados para a construção de aplicações para a plataforma móvel iOS:

- DCCIOS32: compila aplicações iOS que irão rodar no iOS Simulator no Mac OS X (opção iOS Simulator de Target Platforms);

• DCCIOSARM: compila aplicações iOS que irão rodar em um dispositivo iOS (opção iOS Device de Target Platforms);

Sendo assim, eles ficam disponíveis exclusivamente para projetos FireMonkey Mobile Application. Em termos de evolução, a inserção destes novos compiladores Mobile no contexto do produto dá início à "próxima geração" de compiladores Delphi, em adição aos compiladores existentes para Desktop – DCC32 (Win32), DCC64 (Win64) e DCCOSX (Mac OS X). Tecnicamente falando, os compiladores Mobile apresentam algumas diferenças pontuais com relação ao compiladores Desktop. A seguir seguem algumas de suas características peculiares:

• Sem suporte a COM;
• Sem suporte a sete tipos de string já conhecidos – AnsiString, WideString, AnsiChar, PAnsiChar, PWideChar, OpenString e ShortString;
• Strings no iOS são baseadas em 0 (0-based) ao invés de 1 (1-based), o que deve ser levado em conta durante a manipulação destes tipos;
• Contagem de referência automática (Automatic Reference Counting).

Automatic Reference Counting (ARC)

Trazido como uma das particularidades dos novos compiladores Mobile, Automatic Reference Counting, ou simplesmente ARC, pode ser definido como sendo um recurso que provê uma nova forma de se gerir a vida de um determinado objeto. O seu grande diferencial fica por conta da não mais necessidade de se liberar explicitamente o objeto da memória quando este já não é mais necessário. Isso vai de encontro ao que todo desenvolvedor Delphi estava habituado a fazer utilizando compiladores Desktop, tal como o DCC32, conforme mostra o trecho de código da **Listagem 1**.

Listagem 1. Exemplo de manipulação de objeto (criação e destruição)

```
01 var
02   Objeto1: TPessoa;
03 begin
04   Objeto1 := TPessoa.Create; // cria-se o objeto
05   try
06    Objeto1.DizOlaMundo; // uso do objeto
07   finally
08    Objeto1.Free; // libera o objeto da memória após seu uso
09   end;
10 end;
```

Portanto, este mesmo código apresentado na **Listagem 1**, se compilado com os novos compiladores Mobile, não necessitaria da instrução Objeto1.Free (linha 08). Isto porque, esta mesma ação já seria tratada internamente pelo recurso de ARC, na declaração "end" final, ou seja, no momento em que a variável Objeto1 deixa de fazer parte do escopo atual. Este tipo de mudança acaba, inevitavelmente, por afetar diretamente o estilo de codificação durante o desenvolvimento. Num exemplo simples, para um projeto FireMonkey Mobile Application, o mesmo código apresentado poderia ser reescrito conforme a **Listagem 2**, já visando tirar proveito do recurso de ARC.

Listagem 2. Código utilizando o recurso de ARC

```
01 var
02   Objeto1: TPessoa;
03 begin
04   Objeto1 := TPessoa.Create;
05   Objeto1.DizOlaMundo;
06 end;
```

Nesta nova codificação apresentada na **Listagem 2**, o objeto Objeto1 seria então imediatamente destruído ao final do bloco, quando a variável sai do escopo. Seguindo esta linha de raciocínio, é possível também se antecipar a destruição do objeto a qualquer momento, apenas informando ao compilador que ele não será mais utilizado, tal como atribuindo-lhe "nil", o que faz com que seja zerada a contagem de referência do objeto. Exemplo:

```
Objeto1 := nil;
```

Ainda em função do recurso de ARC provido pelos novos compiladores, é habilitado para os projetos que fazem uso deles uma nova propriedade somente leitura, do tipo Integer, denominada RefCount. Esta propriedade é declarada na nova classe TAutoObject (que herda de TObject) e retorna, em números, a contagem de referência de um objeto. Caso se tente utilizá-la com compiladores Desktop, tal como num projeto VCL Forms (Win32), uma exceção é gerada em tempo de compilação, conforme mostra a **Figura 9**.

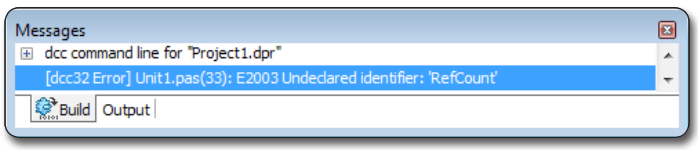

Figura 9. Exceção levantada ao se tentar fazer uso da propriedade RefCount

Vale lembrar que o recurso de contagem de referência não é totalmente novo no Delphi, uma vez que o mesmo já provê este tipo de funcionalidade para strings e objetos referenciados através de variáveis de algum tipo Interface.

FireMonkey Platform Services

No Delphi XE4, o framework FireMonkey traz ainda um novo recurso chamado Platform Services, útil para o desenvolvedor se certificar de que a plataforma de destino selecionada para o aplicativo produzido irá realmente suportar um determinado serviço. Assim, é possível evitar eventuais problemas em tempo de execução, tal como o uso de um recurso não disponível pela plataforma utilizada. Na prática, o Platform Services pode ser definido como sendo um registro de vários serviços de plataforma (platform service) e que é implementado por uma classe do framework chamada TPlatformServices, declarada na unit FMX.Platform. Dito isto, esta classe já dispõe de métodos úteis à sua funcionalidade, tais como para consultar se um serviço é suportado (SupportsPlatformService), bem como para se adicionar (AddPlatformService) ou remover (RemovePlatformService) um determinado serviço do registro.

Ainda neste contexto, um serviço de plataforma (platform service) nada mais é do que uma interface que especifica uma funcionalidade que pode ou não ser implementada em uma determinada plataforma em Runtime. Neste estágio atual do FireMonkey, a lista de interfaces de serviços disponíveis inclui elementos como:

- IFMXClipboardService: interface que define o serviço de Clipboard, ou seja, as funções relacionadas à definição e obtenção de valores da área de transferência na plataforma;
- IFMXCursorService: interface que disponibiliza métodos relacionados às tratativas do cursor (ponteiro) do mouse;
- IFMXDragDropService: interface que define o serviço de Drag & Drop, que envolve as funcionalidades relacionadas à operação de arrastar e soltar;
- IFMXLocaleService: interface que define o serviço de Locale, ou seja, funcionalidades relacionadas à localidade atual da plataforma do aplicativo, tal como a obtenção de informações como o identificador de seu idioma atual;
- IFMXSystemFontService: interface que define o serviço de Font, ou seja, a funcionalidade necessária para obtenção da família da fonte padrão da plataforma.

Já olhando pelo lado do desenvolvimento efetivo, uma boa prática ao se fazer uso de determinado serviço na aplicação é verificar se ele realmente é suportado pela plataforma onde será executada. Sendo assim, em vista do já comentado método SupportsPlatformaService disponibilizado pela classe TPlatformServices, a **Listagem 3** mostra um exemplo desta verificação com o serviço IFMXCursorService.

Listagem 3. Exemplo de verificação com o serviço IFMXCursorService

```
if TPlatformServices.Current.SupportsPlatformService(IFMXCursorService) then
```

Compilação – Novos símbolos condicionais

No Delphi, um símbolo condicional (conditional symbol) pode ser definido como sendo uma espécie de variável booleana que trabalha com dois estados: defined (True) e undefined (False). Cada um destes símbolos atua então como uma diretriz para o compilador, estabelecendo uma compilação condicional (conditional compilation) de uma aplicação. Cada símbolo existente é avaliado durante o processo de compilação do código-fonte do programa. Num exemplo simples, em uma aplicação do tipo Console Application, nota-se a presença do símbolo CONSOLE, que indica ao compilador que a aplicação corrente deverá ser compilada como uma aplicação console. Da mesma forma, o símbolo WIN32 indica que a plataforma de destino é Windows 32-bits.

Com o XE4 são introduzidos novos símbolos condicionais, num total de nove, muitos dos quais definidos apenas para os novos compiladores para iOS (DCCIOS32 e DCCIOSARM):

- AUTOREFCOUNT;
- CPUARM;
- EXTERNAL_LINKER;
- IOS;
- NEXTGEN;
- UNDERSCOREIMPORTNAME;
- WEAKREF;
- WEAKINSTREF;
- WEAKINTREF.

IBLite for iOS

O InterBase ToGo é a edição embedded (embarcada) do SGBD InterBase da Embarcadero. Diz-se embarcado pelo fato dele ser uma engine de banco de dados que pode ser diretamente "embutida" em aplicações, através do uso da biblioteca do próprio InterBase. Isso faz com que o InterBase ToGo não precise ser instalado (seja no servidor ou na máquina cliente), o que facilita uma eventual distribuição da própria aplicação. Ele é segmentado ainda em duas plataformas diferentes, sendo a própria ToGo e

outra denominada IBLite. Pelo aspecto técnico, ambas apresentam o mesmo modelo de desenvolvimento, sendo que a primeira possui um custo para Deploy enquanto a segunda não. Com a incorporação do suporte à iOS no XE4, tanto ToGo quanto IBLite podem ser utilizados em aplicações para esta plataforma.

FireDAC

FireDAC é a mais recente biblioteca de componentes de acesso a dados incorporada ao Delphi. Tal como o dbExpress (**BOX 4**), uma de suas principais características é prover acesso direto a diversas fontes de dados, que vão desde tradicionais SGBDs (Firebird, InterBase, MySQL, SQL Server, Oracle e DB2) a servidores DataSnap e conexões ODBC. Incorporado ainda na versão XE3 (sendo tema de artigo aqui mesmo nesta revista), o FireDAC surge agora efetivamente como uma das novidades do Delphi XE4. O grande diferencial neste lançamento é que ele pode ser usado na construção de aplicações para Windows, Mac OS X e iOS. Apesar de não fazer parte da instalação principal do IDE, ele é disponibilizado de forma gratuita para os proprietários das edições Enterprise, Ultimate e Architect do Delphi.

BOX 4. dbExpress

dbExpress é um dos frameworks de acesso a dados mais antigos e utilizados do Delphi. Em sua composição, apresenta um conjunto de classes e componentes que fornecem acesso rápido a diversas fontes de dados, incluindo banco de dados e servidores DataSnap. Para cada fonte suportada, o framework fornece um driver específico.

Em razão de sua extensão e por questões de espaço, uma lista completa das novidades trazidas pela versão XE4 do Delphi pode ser vista no endereço disponibilizado na seção **Links**.

Conclusão

Este artigo teve como tema central a nova versão do Delphi, a XE4, e a apresentação de suas principais novidades. Conforme já é de costume, este novo lançamento da Embarcadero trouxe mais uma vez uma série de novos recursos e melhorias em diversas partes do contexto Delphi, no entanto, com as atenções maiores voltadas ao desenvolvimento para dispositivos móveis (iOS). Aliás, este parece ser (e já vem sendo) a maior e mais recente aposta da empresa para as novas diretrizes comerciais do Delphi. Em meio à comunidade, as opiniões se divergem sobre o assunto. Em território nacional, é certo que uma boa parcela dos desenvolvedores ainda não tem os dispositivos Apple (iPhone, iPad e Mac) inseridos em sua realidade comercial/profissional. Ao mesmo tempo, a Embarcadero dá cada vez mais mostras de sua preocupação em evoluir a ferramenta, incorporando novos recursos e abrindo ainda mais o leque de opções para o desenvolvedor.

Fato é que agora na versão XE4 do Delphi, a partir de uma mesma base de código é possível se produzir aplicativos para múltiplas plataformas (Windows, Mac OS X e iOS) e múltiplos dispositivos (PCs, Smartphones e Tablets). O desafio mais uma vez está lançado e cabe a você redefinir (ou não) as suas diretrizes de desenvolvimento.

Agradeço a todos e até a próxima.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links e Referências:

Delphi XE4 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE4 – Página oficial do produto
http://www.embarcadero.com/products/delphi

Delphi XE4 – What's News in Delphi and C++ Builder XE4
http://docwiki.embarcadero.com/RADStudio/XE4/en/What%27s_New_in_Delphi_and_C%2B%2BBuilder_XE4

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Aplicando boas práticas para criação de objetos

Padrões de projeto em Delphi

Resumo DevMan

Aplicando boas práticas para criação de objetos:

Softwares reais evoluem. Eles são constantemente modificados e adaptados, novos recursos são implementados, bugs são corrigidos, outros criados e aparecem do nada, novas exigências são feitas, a lei muda, o cliente pede, o código vai se tornando mais complexo e se afasta do seu projeto original, diminuindo sua qualidade. Está mais do que óbvio que a maior parte do custo total do desenvolvimento do software é dedicada à sua manutenção, em alguns casos, esse valor pode chegar a 80% do seu custo final. Se uma aplicação precisa ser modificada em um determinado momento, seja para manutenção corretiva ou evolutiva para atender novas funcionalidades, este processo precisa ser feito de forma disciplinada, correta, usando padrões, já que o custo da manutenção representa um porcentual elevado comparado ao custo total do sistema, como citado.

Com isso, o uso de padrões de projeto, ou Design Patterns, torna um software mais fácil de ser mantido em longo prazo, reduzindo custos ou otimizando o retorno do investimento, apesar de exigir um maior esforço em um primeiro momento, durante o design do projeto a ser realizado. Como muitas empresas não realizam este projeto da forma correta na concepção do software, seja pela falta de tempo devido a prazos reduzidos, desconhecimento da equipe sobre uso de boas práticas de programação orientada a abjetos, arquitetura ou uso de padrões, execução de múltiplos projetos simultaneamente, ocorre o chamado débito de projeto. São softwares que atendem aos seus requisitos em um estágio inicial, mas são extremamente rígidos em sua estrutura, difíceis de manter e evoluir, repleto de "bad smells", principalmente se essa evolução for feita por outro desenvolvedor que não o próprio criador. Um bad smell, ou simplesmente mau cheiro, é uma indicação superficial que leva a um problema profundo no código de um sistema. O termo foi criado por Kent Beck e utilizado no livro Refactoring: Improving the Design of Existing Programs de Martin Fowler [2].

Este artigo apresenta os fundamentos sobre Padrões de Projeto e implementações de exemplos práticos reais em Delphi. Neste artigo, serão tratados os principais padrões criacionais do GoF [1], destinados a resolver problemas com criação de objetos concretos, que comprometem a reutilização de módulos do sistema e aumentam o acoplamento. Os padrões apresentados neste artigo são o Abstract Factory, Factory Method, Singleton e Builder.

Em que situação o tema é útil:

Padrões de projeto são extremamente úteis para criar softwares mais fáceis de serem mantidos, evoluídos, promovendo reutilização de código, uso de boas práticas de programação orientada a objetos, redução de custos com manutenção, já que permitem que alterações em um determinado sistema de software não quebrem outras funcionalidades, reduzindo a dependência entre classes, através do uso exaustivo de abstrações. Dessa forma, são úteis para serem aplicados em projetos de média e larga escala.

O conhecimento sobre padrões de projeto só faz sentido para aqueles que desenvolvem softwares de forma orientada a objetos. Não faz sentido tentar aplicar padrões de projeto se você programa centenas de linhas de código em eventos de um controle de user interface em um formulário Delphi, sem separar responsabilidades. Não faz sentido tentar usar padrões se você usa event-driven programming (programação dirigida ou orientada a eventos).

Se o software feito em Delphi da sua empresa precisa passar por uma evolução, modernização, migração, porém está mal escrito, repleto de bad smells, onde dezenas de programadores já colocaram a mão, e o custo para mantê-lo está saindo caro, talvez a melhor atividade antes de aplicar padrões de projeto seja a refatoração. Aplicar refatorações primitivas, como extrair métodos, classes, superclasses, pode tornar o seu software mais modular, simples de manter e evoluir. A partir daí, o uso de padrões de projeto pode ajudar a melhorar os atributos de qualidade do seu software.

Uma outra atividade, muito interessante, é aplicar padrões de projeto no contexto de refatorações (**BOX 1**).

Ou seja, nunca se aplica um padrão de forma antecipada, pois não sabe se realmente ele será necessário, causando uma complexidade desnecessária, o que chamamos de excesso de engenharia. Programadores não são videntes. Em contrapartida, nunca aplicar um padrão causa uma escassez de engenharia muito comum em aplicações que seguem o modelo RAD de desenvolvimento, como Delphi (existem programadores que usam Delphi há mais de 15 anos e nunca criaram uma classe). O software já tem 10 anos, está na versão 5, ninguém mais entende como ele funciona, o que foi feito, as pessoas perdem a credibilidade, a motivação, os erros são muitos, uma nova funcionalidade que deveria ser implementada em minutos leva horas ou dias, a correção de um bug gera novos bugs, a rotatividade de membros na equipe de desenvolvimento da empresa está alta, e isso está cada vez pior. Tão ruim que o caos já está generalizado e a empresa decide em algum momento reescrever o software do zero, pois não há mais conserto. Nesse sentido, a prática de aplicar padrões de projeto em código existente, no contexto de uma refatoração, se torna uma atividade bastante atraente.

BOX1. Refatoração

A refatoração (do inglês Refactoring) é o processo de modificação do sistema de software sem alterar seu comportamento externo observável, como funcionalidades, tendo como alguns dos seus benefícios melhorar o entendimento do código, remover duplicidades, simplificar métodos, facilitando assim a manutenção, correção de bugs, adição de novas funcionalidades, aumentando a qualidade do produto [3].

Padrões de projeto se baseiam fortemente em princípios básicos da programação orientada a objetos, mais especificamente, na herança, abstração, encapsulamento e polimorfismo. Aqui é importante distinguir entre herança de classe da herança de abstrações. As heranças que veremos nos padrões são todas de abstrações, de interfaces, sem implementação. É uma herança de apenas dois níveis, já que criar cadeias muito grandes de hierarquia causa complexidade e dificulta a manutenção. Nesse ponto, uma herança de "caixa preta" se torna mais atraente, como faz o padrão Decorator. Se não há o domínio estes conceitos, principalmente a implementação de classes abstratas por polimorfismo, é recomendado fortemente que faça uma pesquisa e se domine estes fundamentos antes de entrar de cabeça no mundo dos padrões.

Os padrões discutidos neste artigo são apresentados no famoso livro "Design Patterns: Elements of Reusable Object-Oriented Software", escrito em 1995 por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides [1]. Eles ficaram conhecidos como os padrões GoF (Ganf of Four, ou "gangue dos quatro"), em alusão aos seus autores. Ao todo são 23 padrões de projeto, divididos em três categorias: criacionais, estruturais e comportamentais. Este artigo aborda os principais padrões do grupo criacionais.

Padrões de projeto são largamente utilizados no próprio Delphi. Apesar do Delphi ter sido desenvolvido antes do lançamento do livro (o Delphi foi criado a partir do Borland Pascal for Windows na década de 90, que veio do Turbo Pacal da década de 80), pode-se notar muitos padrões de projeto na própria VCL do Delphi e no IDE. Por exemplo, a VCL inteira não deixa de ser uma grande "fachada" para esconder a complexidade e total deselegância da implementação procedural da API do Windows baseada em ponteiros sem tipo e sem classes. É a mais clara implementação que se pode ter do padrão Facade, assim como o próprio .NET Framework também é. Isso porque esses frameworks encapsulam um subsistema inteiro tornando-o mais fácil de utilizar. Quando olhamos para a parte visual da VCL, como os controles, formulários, botões, panels, e o modelo de componente (TComponent), vemos a implementação clara do padrão Composite (composição). De fato, podemos tratar um formulário como uma composição de outros componentes, enquanto o próprio TForm também é tratado como um TComponent (pois em algum momento herda dele). Um DataSource, ao apontar para um DataSet, usa abstração para esconder um tipo concreto e resolve uma chamada Post de forma polimórfica, a fim de usar uma das n diferentes formas de persistir dados em um banco de dados a partir de dados vindos da tela. Ele esconde uma estratégia para persistir os dados, de fato, você pode trocar um TTable por um TClientDataSet configurados da forma correta e os dados serão gravados no mesmo local. É uma espécie de padrão Strategy. Existem objetos na VCL que podem ser utilizadas uma única vez, como Application, por exemplo, o que poderia caracterizar um Singleton. Quando chamamos um método Post de qualquer implementação de um TDataSet, inúmeros outros métodos polimórficos são chamados em TDataSet mas que só são conhecidos em classes descendentes. Eles só estão em TDataSet para serem chamados de acordo com um determinada ordem, um típico Template Method. E por aí vai, existem dezenas de padrões implementados na VCL (e FireMonkey), comprovando que mesmo na década de 90 a equipe de desenvolvimento da então Borland tinha fortes conhecimentos em orientação a objetos. Essas soluções, não só as utilizadas pela equipe de desenvolvimento da Borland, se tornaram tão claras, que foram documentadas para que pudessem ser reutilizadas em outros projetos, como receitas de bolo prontas. Nascia o livro Padrões de Projeto do GoF.

Antes de iniciarmos a apresentação dos padrões criacionais do artigo, vamos comentar como estes exemplos serão apresentados. Primeiramente, para cada padrão, eu apresentarei a descrição formal, sua intenção e os participantes do padrão, segundo as referências do livro do GoF [1]. A seguir, depois de conhecidos os fundamentos, são apresentados para cada padrão um exemplo prático, real (não usarei exemplos fictícios com animais, cachorros ou formas geométricas). Diagramas de classes UML, criados no próprio Delphi através do Model Support, ajudarão a compreender melhor cada padrão. Os exemplos são todos feitos em aplicações de Console, porque aqui a interface não importa, de fato, padrões podem ser usados em VCL, FireMonkey, ASP .NET, WPF, Windows Forms etc.

Abstract Factory

O padrão Abstract Factory, ou "fábrica abstrata", fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas (Gamma et al., 1995). Nesse padrão, a fábrica (conhecida como Factory) recebe solicitações por objetos concretos a partir de um objeto cliente. Este padrão pode ser utilizado quando um sistema de software precisa ser independente de como classes concretas (chamadas de produtos) são criadas. A fábrica fica responsável por encapsular a criação de uma família de produtos. Um objeto cliente precisa conhecer somente as interfaces desses produtos, não a sua implementação, aumentando o encapsulamento e abstração. A aplicação deste padrão traz como benefício principal o isolamento de classes concretas (produtos). Se for necessário trocar uma família inteira de produtos, esse processo se torna menos impactante nas demais partes do sistema, pois os produtos ficam isolados e não expõem sua implementação a classes clientes, diminuindo o acoplamento.

Os participantes do padrão são os seguintes:

- Abstract Factory – declara uma interface abstrata para operações que criam objetos-produtos;
- Concrete Factory – implementa as operações que criam objetos-produtos concretos;
- Abstract Product – declara uma interface para um tipo de objeto-produto;
- Concrete Product – define um objeto-produto a ser criado pela correspondente fábrica concreta; Implementa a interface de Abstract Product;
- Cliente – usa somente as interfaces declaradas pelas classes Abstract Factory e Abstract Product.

Na **Listagem 1** vemos um exemplo de uso do Abstract Factory em Delphi, cujas classes podem ser vistas no diagrama de classes UML da **Figura 1**. Consiste de um framework para acesso a dois bancos de dados distintos, Oracle e SQL Server. No código defini uma classe abstrata para representar conexões a banco de dados, chamada de DBConnection, que tem um método abstrato chamado Open. Duas classes descendentes (o que chamamos de objetos-concretos ou produtos) implementam essa abstração para os dois tipos de bancos de dados, OracleConnection e SqlConnection. Usando polimorfismo, essas classes concretas dão um override em um método virtual abstract apenas definido na classe base (abstrata). Da mesma forma, existe uma abstração para execução de comandos no banco de dados, representada aqui pela classe DBCommand, que também por herança e polimorfismo tomam duas representações concretas, uma para cada tipo de banco, são as classes OracleCommand e SqlCommand. Estes produtos implementam de forma concreta um método chamado Execute, que processa a execução de um comando SQL no banco dados. Por fins didáticos, o foco não está na implementação dessa comunicação com o BD, então apenas é exibida uma mensagem em tela para informar que o método foi chamado. Nosso trabalho aqui é o do projetista, aquele que escolhe e define qual padrão vai ser usado, não do implementador, que se concentra em trabalhar entre os begins e ends.

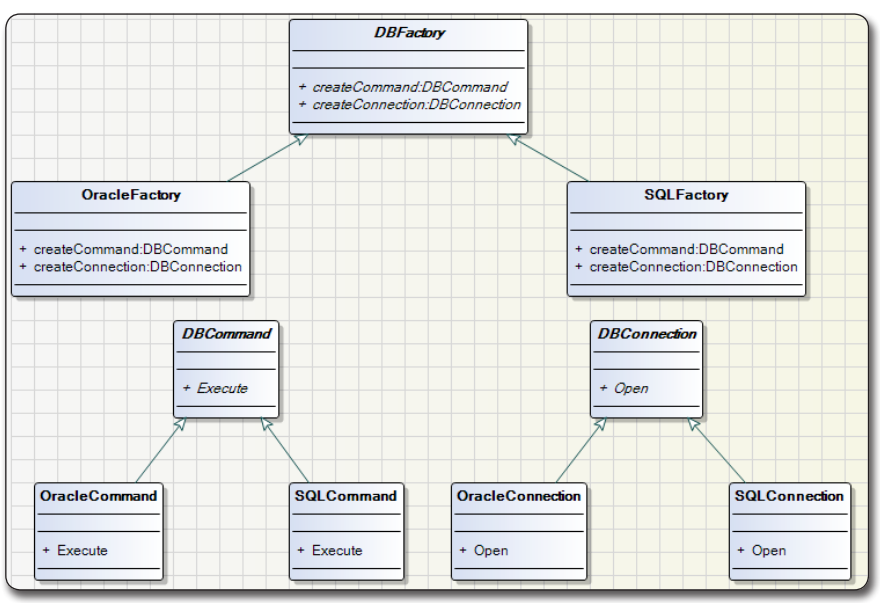

Figura 1. Estrutura de classes do exemplo usando Abstract Factory

Sem usar padrões ou fábricas, normalmente a maioria dos desenvolvedores utilizaria as classes concretas sem pensar em abstrações. Por exemplo, se fosse necessário criar uma conexão e executar um comando no SQL Server, teríamos o código da **Listagem 2.**

Não há problema algum, em curto prazo, utilizar esta abordagem. Afinal, programadores não são videntes, não sabemos se a empresa que nos contratou decidirá um dia por utilizar o banco de dados Oracle ao invés do SQL Server. O problema é que utilizar classes concretas causa uma dependência muito grande entre classes clientes (formulários, por exemplo) e frameworks.

Depender de implementações concretas é um problema, porque tudo muda, essa é uma premissa básica quando se desenvolve software. Certamente hoje não se compra uma TV que não tenha uma entrada HDMI, que permita a mudança de um reprodutor de Blu-ray (por um PS4, por exemplo) sem alterar a interface, o que seria uma tragédia. Imagine ter que enviar um televisor de 52 polegadas de volta para fábrica solicitando que a mesma crie uma nova porta no aparelho para que se possa ligar um novo PS4. Certamente não é dessa forma que funciona na vida real. Na vida real, existem padrões, abstrações e interfaces (nesse caso HDMI), que clientes e fornecedores declaram e implementam para que mudanças não tenham impacto na forma como os objetos são relacionados. Imagine se a especificação da interface USB fosse modificada? Bilhões de dispositivos ao redor do mundo precisariam ser modificados. Na vida real, nós sempre pensamos para abstrações, em qualquer situação,

Listagem 1. Abstract Factory

```
unit uFramework;

interface

type

  // Produto Abstrato (Abtract Product)
  DBCommand = class abstract
    public procedure Execute(); virtual; abstract;
  end;

  // Produto Abstrato (Abstract Product)
  DBConnection = class abstract
    public procedure Open(); virtual; abstract;
  end;

  // Fábrica Abstrata (Abstract Factory)
  DBFactory = class abstract
    public function createConnection(): DBConnection; virtual; abstract;
    public function createCommand(): DBCommand; virtual; abstract;
  end;

  // Produto Concreto (Concrete Product)
  OracleCommand = class(DBCommand)
    public procedure Execute(); override;
  end;

  // Produto Concreto (Concret Product)
  OracleConnection = class(DBConnection)
    public procedure Open(); override;
  end;

  // Produto Concreto (Concrete Product)
  SQLCommand = class(DBCommand)
    public procedure Execute(); override;
  end;

  // Produto Concreto (Concret Product)
  SQLConnection = class(DBConnection)
    public procedure Open(); override;
  end;

  // Fábrica Concreta (Concrete Fabric)
  OracleFactory = class(DBFactory)
    public function createConnection(): DBConnection; override;
    public function createCommand(): DBCommand; override;
  end;

  // Fábrica Concreta (Concrete Fabric)
  SQLFactory = class(DBFactory)
    public function createConnection(): DBConnection; override;
    public function createCommand(): DBCommand; override;
  end;

implementation

procedure OracleCommand.Execute();
begin
  WriteLn('Método Execute de OracleCommand foi chamado');
end;

procedure OracleConnection.Open();
begin
  WriteLn('Método Open de OracleConnection foi chamado');
end;

function OracleFactory.createCommand(): DBCommand;
begin
  result := OracleCommand.Create();
end;

function OracleFactory.createConnection: DBConnection;
begin
  result := OracleConnection.Create();
end;

procedure SQLCommand.Execute();
begin
  WriteLn('Método Execute de SQLCommand foi chamado');
end;

procedure SQLConnection.Open();
begin
  WriteLn('Método Open de SQLConnection foi chamado');
end;

function SQLFactory.createCommand(): DBCommand;
begin
  result := SQLCommand.Create();
end;

function SQLFactory.createConnection(): DBConnection;
begin
  result := SQLConnection.Create();
end;

end.
```

pare para pensar e reflita. Se eu pretendo trocar de carro em 2015, eu não digo "vou comprar o modelo XYZ Sport 7.2" e ponto final. Pode ser que o modelo saia de linha, fique muito caro, ou um novo modelo mais moderno o substitua. Eu simplesmente falo que vou trocar de carro e pronto, pois "carro" é uma abstração do modelo, e com isso eu não quebro meu contrato. Fico preparado para mudanças. Assim deveria ser no seu sistema.

Da mesma forma, o padrão Abstract Factory promove uma das melhores práticas da orientação a objeto, conhecida como

Listagem 2. Código sem abstração

```
var
  con: SqlConnection;
  cmd: SqlCommand;
begin
  con := SqlConnection.Create();
  con.Open();
  cmd := SqlCommand.Create();
  cmd.Execute();
...
```

"Programe para uma interface / abstração". Ao invés de consumirmos diretamente as implementações concretas (exemplo anterior), lidamos com as suas abstrações. Mais precisamente, nunca deveríamos utilizar um Create diretamente em uma classe que pode um dia mudar, ou ainda, ser totalmente substituída por estar obsoleta, descontinuada ou com problema. Que tal pensar na substituição do BDE para DBX ou ainda FireDac? Usar Abstract Factory resolveria essa migração com a alteração de apenas uma linha de código, ou até nenhuma, se fosse utilizado um arquivo de configuração.

Ao invés de criar classes de forma concreta, o padrão Abstract Factory sugere que se solicite isso a uma fábrica, que também possui uma abstração. Isso significa que há uma definição virtual de como conexões e comandos devem ser criados, definidas aqui pela classe DBFactory. Existem dois métodos polimórficos virtuais, chamados createCommand e createConnection. É através destes métodos virtuais que classes clientes devem solicitar objetos concretos, nunca usando Create diretamente. Também por herança e polimorfismo, a fábrica abstrata é feita concreta através da implementação de OracleFactory e SqlFactory. Por exemplo, createCommand da fábrica abstrata por definição retorna um DBCommand, que também é uma abstração. É o SqlFactory que sobrescreve createCommand para dizer que ela sabe qual é o produto concreto real a ser devolvido pela função, nesse caso, um SqlCommand. Isso libera classes clientes de conhecerem quais os tipos concretos uma fábrica constrói, reduzindo o acoplamento. Se o cliente quiser mudar todos os tipos concretos para conexão e execução de comandos no banco de dados, basta trocar de fábrica, e nada mais. Todas as demais declarações e chamadas são resolvidas, sem qualquer alteração no código, por abstração e polimorfismo. Nada de IFs para testar tipos, jamais. Polimorfismo puro.

A **Listagem 3** mostra isso funcionando na prática. Note que estamos declarando na sessão var somente abstrações, incluindo uma referência para uma fábrica abstrata. A primeira linha do código é quem coloca dentro dessa referência uma implementação concreta (produto) da fábrica: `db := OracleFactory.Create()`. Executando a aplicação, veremos que tanto a conexão quanto o comando foram executados no banco de dados Oracle. Agora, e se for necessário modificar todo o acesso a dados do sistema, para conectar e executar comandos no SQL Server? Basta trocar a primeira linha por `db := SqlFactory.Create()`. Pronto, executando a aplicação, as instâncias utilizadas serão as concretas (SqlCommand e SqlConnection), porém as declarações abstratas e chamadas de métodos permancem intactas, tudo resolvido por polimorfismo.

Nota

Uma opção seria colocar o nome da classe da fábrica concreta em um arquivo de configuração XML ou INI, e usando reflexão (RTTI no caso do Delphi), ler esta classe dinamicamente para então criar a fábrica em tempo de execução. Com isso seria possível alterar todo o framework (família) de classes de acesso a dados, de Oracle para SQL Server, por exemplo, sem mesmo recompilar a aplicação.

Listagem 3. Usando o Abstract Factory

```
program AbstractFactory;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  db: DBFactory;
  con: DBConnection;
  cmd: DBCommand;
begin
  db := OracleFactory.Create();
  con := db.createConnection();
  con.Open();
  cmd := db.createCommand();
  cmd.Execute();
  ReadLn;
end.
```

Factory Method

O padrão Factory Method, ou "Método de Fábrica", tem a intenção de definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. Factory Method permite adiar a instanciação para subclasses (Gamma et al., 1995). O padrão pode ser aplicado quando uma classe não é capaz de saber antecipadamente quais objetos devem realmente ser criados, ou ainda, quando uma classe repassa (delega) para suas subclasses a responsabilidades de quais objetos realmente devem ser criados. O uso do padrão traz vários benefícios, sendo o maior deles a flexibilidade. Como a criação de um objeto é encapsulada por um método de fábrica, classes clientes não criam esses objetos diretamente. Além disso, como uma classe cliente usa o método de fábrica para obter uma instância de uma classe desejada, o padrão pode ser usado como uma forma transparente e flexível para fornecer um objeto diferente. Isso porque é possível retornar uma subclasse compatível com a classe do objeto retornado pelo método.

Os participantes do padrão são os seguintes:

- Product - define a interface de objetos que o método de fábrica cria;
- Concrete Product – implementa a interface Product;
- Creator – Declara o método de fábrica, o qual retorna um objeto do tipo Product;
- Concrete Creator – redefine o método fábrica para retornar uma instância de um Concrete Product.

O Factory Method se assemelha muito ao padrão Abstract Factory. A diferença básica é que, enquanto o Abstract Factory permite substituir uma família inteira de classes, por exemplo, um framework de acesso a dados ou de interface gráfica, o Factory Method permite variar apenas um tipo de produto concreto. Um Factory Method é basicamente um construtor com poli-

morfismo, um construtor virtual. Pede-se a uma classe abstrata (Creator) para que crie um produto baseado em sua abstração. Por polimorfismo uma classe descendente (Concrete Creator) dá um override no método de construção (o Factory Method) e informa qual é o produto concreto que deve ser realmente criado, no local da abstração. Como o Creator abstrato devolve um tipo base de produto, então um Concrete Creator pode devolver uma instância compatível (por herança) com o tipo mais primitivo devolvido pelo Factory Method virtual da classe base abstrata.

Com isso, há uma redução muito grande do acoplamento e dependência entre as classes. Ao invés de testar com IFs que tipo de estrutura deve ser criada, o que é um erro, deixa-se a cargo do compilador e do polimorfismo resolver em tempo de execução que objeto real vai ser criado. Além disso, torna-se muito fácil adicionar novos produtos compatíveis com a abstração, de forma a substituir um objeto que esteja desempenhado mal sua função no sistema, por exemplo, um componente para envio de e-mails ou um componente para autenticação de usuários.

Imagine que uma aplicação Web precisa agora implementar uma nova forma de autenticação. Ao invés de criar um mecanismo próprio para Login, a empresa decide que a partir deste momento os usuários podem usar suas contas do Facebook ou do Google para logar no sistema, sem criar uma conta diferente no site. Tal abordagem tem sido muito utilizada em sites atualmente. Em um primeiro momento, a empresa decide implementar a autenticação via Facebook. Um programador sem conhecimentos de OO e padrões de projeto, habitualmente criaria a classe para autenticar e faria o consumo da mesma diretamente, sem se preocupar com dependências ou futuras novas implementações de autenticação, como mostra o trecho a seguir:

```
var
  aut: AutenticacaoFacebook;
begin
  aut := AutenticacaoFacebook.Create(); // dependência direta
  aut.Autenticar();
```

O grande problema, novamente, está na criação direta da classe concreta, ferindo novamente o princípio básico de programação para abstrações. Mas como programadores não podem prever o futuro, não se sabe se um dia realmente vai ser necessário implementar um novo tipo de autenticação, pois o requisito inicial dizia somente Facebook. E a mudança vem. Um programador sem conhecimentos de OO e padrões então criaria uma nova classe para autenticar via Google e, sem usar herança nem polimorfismo, poderia fazer algo como na **Listagem 4.** Nela podemos observar, em poucas linhas de código, as piores práticas de programação:

• Código duplicado, nesse caso, a chamada ao método Autenticar;

• Teste de tipo: jamais se deve testar "tipos códigos" em tela e resolver isso através da criação de classes sem herança. Se um novo tipo for necessário, mais um teste deve ser incluído. Além disso, testes tendem a duplicar código e tornar a lógica complexa, o que é um bad smell;

Listagem 4. As piores práticas de programação

```
var
  Face: AutenticacaoFacebook;
  Google: AutenticacaoGoogle;
begin
  if (tipoSelecionadoEmTela = 1) then
  begin
    Face := AutenticacaoFacebook.Create(); // dependência direta
    Face.Autenticar();
  end
  else
  if (tipoSelecionadoEmTela = 2) then
  begin
    Google:= AutenticacaoGoogle.Create(); // dependência direta
    Google.Autenticar();
  end;
...
```

• Há um outro bad smell no código, um "Magic Number", no caso os números "1" e "2". Para outro programador que lê o código, dificilmente saberá o que significam esses números. É um índice de menu? Índice de seleção de um combo? Note que no exemplo simulei uma entrada de tela de usuário;

• Dependência direta de classes concretas, causando forte acoplamento e rigidez no código.

Agora observe o framework apresentado na **Figura 2** e **Listagem 5**. É a implementação do padrão Factory Method. Note que o primeiro passo, supondo que já tenha criado uma classe de autenticação, é aplicar uma refatoração Extract Super Class, movendo os métodos que variam para a abstração de nível superior, criando-se assim uma relação de herança entre AutenticacaoFacebook (produto concreto) e Autenticacao (abstração). Para criar estes produtos,

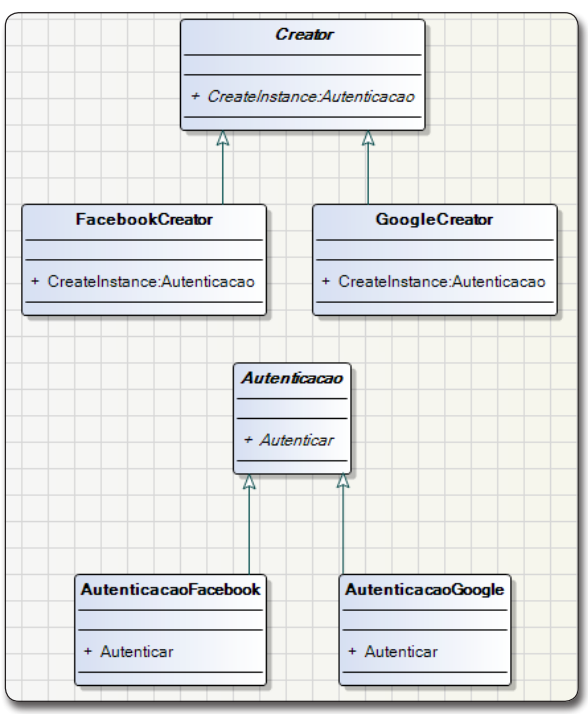

Figura 2. Estrutura de classes do exemplo usando Factory Method

existe uma classe chamada Creator, que tem um método polimórfico (o nosso Factory Method) chamado CreateInstance, que devolve uma abstração (Autenticacao). Por polimorfismo, duas classes descendentes, chamadas aqui de FacebookCreator e GoogleCreator sobrescrevem (override) este método e devolvem um produto concreto, que por herança, é compatível com o produto abstrato, logo não causando um erro na verificação de tipos pelo compilador do Delphi.

Listagem 5. Factory Method

```
unit uFramework;

interface

type
 // Product
 Autenticacao = class abstract
  public procedure Autenticar(); virtual; abstract;
 end;

 // Produto Concreto
 AutenticacaoFacebook = class(Autenticacao)
  public procedure Autenticar(); override;
 end;

 // Produto Concreto
 AutenticacaoGoogle = class(Autenticacao)
  public procedure Autenticar(); override;
 end;

 // Creator - define o Factory Method abstrato
 Creator = class abstract
  public function CreateInstance(): Autenticacao; virtual; abstract;
 end;

 // Concrete Creator
 FacebookCreator = class(Creator)
  public function CreateInstance(): Autenticacao; override;
 end;

 // Concrete Creator
 GoogleCreator = class(Creator)
  public function CreateInstance(): Autenticacao; override;
 end;

implementation

procedure AutenticacaoFacebook.Autenticar();
begin
 WriteLn('Autenticado via Facebook');
end;

procedure AutenticacaoGoogle.Autenticar();
begin
 WriteLn('Autenticado via Google');
end;

function FacebookCreator.CreateInstance(): Autenticacao;
begin
 result := AutenticacaoFacebook.Create();
end;

function GoogleCreator.CreateInstance(): Autenticacao;
begin
 result := AutenticacaoGoogle.Create();
end;

end.
```

A **Listagem 6** mostra como consumir as classes do framework. No exemplo, foi definida uma lista genérica de Creators (criadores), que é uma abstração para criação de produtos. São adicionados nessa coleção então dois criadores concretos, um para o Facebook e outro para o Google. O objeto que representa uma autenticação é uma abstração, chamada aqui de "aut". Então, um for in (o foreach do Delphi) percorre a coleção chamando o Factory Method de cada criador, chamado CreateInstance, que por polimorfismo vai colocar na referência "aut" um tipo concreto (AutenticacaoFacebook ou AutenticacaoGoogle) compatível. Note que não há teste algum com IF, e a chamada ao método Autenticar também é resolvida por polimorfismo, tanto que aparece uma única vez no código, sem redundância. O exemplo, quando executado, fará a simulação da autenticação do usuário nos dois serviços citados, já que usamos uma coleção. Agora observe, se um novo tipo de autenticação for necessário (Twitter), o que é afetado? Praticamente nada, você criará o produto concreto normalmente, o seu creator, e adicionará apenas uma linha ao código que consome estas classes, que já poderá fazer a autenticação usando a nova rede social adicionada.

Listagem 6. Usando o Factory Method

```
program FactoryMethod;

{$APPTYPE CONSOLE}

{$R *.res}

uses
 System.SysUtils,
 Generics.Collections,
 uFramework in 'uFramework.pas';

var
 Creators: TList<Creator>;
 c: Creator;
 aut: Autenticacao;
begin
 Creators := TList<Creator>.Create();
 Creators.Add(FacebookCreator.Create());
 Creators.Add(GoogleCreator.Create());
 for c in Creators do
 begin
  aut := c.CreateInstance(); // Factory Method
  aut.Autenticar();
 end;
 ReadLn;
end.
```

Singleton

O padrão Singleton tem a intenção de garantir que uma classe tenha apenas uma instância, e fornecer um ponto global de acesso a ela (Gamma et al., 1995). O padrão pode ser usado quando for preciso haver apenas uma instância de uma classe, e essa instância tiver que dar acesso aos clientes através de um ponto bem conhecido. Ou quando a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usar uma instância estendida sem alterar o seu código. Entre alguns

benefícios do uso do padrão, podemos citar o acesso controlado à instância única; pode-se criar subclasses do singleton sem impacto nas classes clientes; permite ainda o controle do número de instâncias internamente.

O único participante do padrão é o seguinte:

• Singleton – define uma operação Instance que permite aos clientes acessarem sua única instância. Instance é uma operação de classe (método estático); pode ser responsável pela criação da sua própria instância única.

O Singleton é provavelmente o padrão de projeto mais simples da família GoF, tanto que ele tem um único participante, o próprio Singleton, e nem vai precisar de diagrama UML aqui. A **Listagem 7** mostra a definição de um exemplo, que simula uma classe de conexão com um banco de dados. O Singleton é muito utilizado nesses casos para limitar o número de conexões criadas, já que é um recurso valioso que pode custar caro se mal utilizado (licenças).

Listagem 7. Singleton

```
unit uFramework;

interface

type
 // Singleton
 ConexaoBD = class
  // guarda internamente a instância
  private class var _instance: ConexaoBD;
  // esconde construtor
  protected constructor Create();
  // método que retorna singleton
  public class function Instance(): ConexaoBD;
  public stringConexao: string;
  public procedure Open();
 end;

implementation

constructor ConexaoBD.Create();
begin
end;

class function ConexaoBD.Instance(): ConexaoBD;
begin
 // garantir que sempre uma instância exista, só uma
 if _instance = nil then
   _instance := ConexaoBD.Create();
 result := _instance
end;

procedure ConexaoBD.Open();
begin
 WriteLn('Abrindo conexão com banco ' + stringConexao);
end;

end.
```

Note que o primeiro passo é privatizar o construtor, escondendo-o de forma que outras classes clientes não possam criar instâncias suas através de Create. O outro passo é definir uma variável de classe (estática), que é um atributo privado do tipo da própria classe. Ou seja, a classe guarda internamente a sua própria instância, e não mais o cliente que a utiliza. A seguir temos um método estático (isso se faz em Delphi com a palavra reservada class) que substitui o Create, chamado Instance. É através dele que classes clientes solicitarão a instância do objeto. Note que em sua implementação, é feito um teste para ver se o ponteiro interno privado para a própria classe, que precisa ser protegido, já foi instanciado. Se sim, ele é retornado, senão, ele é criado uma única vez (daí o nome Singleton). Ou seja, não importa quantos clientes utilizarem o Singleton, haverá somente uma instância ativa de conexão com o banco de dados.

Apesar do nome, um Singleton não serve somente para permitir a criação de uma única instância de uma classe. Como você pode ver, através dele nós temos o controle total sobre como o objeto é criado, responsabilidade que não é mais das classes clientes que o consomem. Tudo é feito internamente, de forma privada. Então fica muito fácil, por exemplo, criar o que chamamos de "Object Pooling" (outro padrão, porém não GoF). Ele é uma extensão do Singleton, e é semelhante a outro padrão chamado Prototype. Ao invés de mantermos um único objeto interno, poderíamos criar, por exemplo, 10 objetos e deixá-los instanciados em uma coleção TList<T> interna ou ainda HashTable, ou ainda, ir criando estes por demanda, o que chamamos de "Lazy Loading" (carga tardia). Quando se fala de conexões a banco, isso é extremamente valioso, tanto que este padrão é largamente implementando em frameworks de acesso a dados como o ADO.NET do .NET Framework, o próprio DBExpress e também no Entity Framework. Através do chamado "Connection Pooling" (um tipo de Object Pooling), esses frameworks conseguem manter uma coleção reutilizável de componentes de conexão reciclados. Ao invés de criar e destruir várias conexões a todo o momento para cada cliente que solicita uma operação no BD, o framework cria um pool de objetos reciclados que não são destruídos depois de utilizados, podendo ser reaproveitados, aumentando a performance das soluções.

A **Listagem 8** mostra como consumir o Singleton. Note que define duas variáveis de conexão, e usei o Instance do Singleton para devolver a instância. Internamente, tanto "con1" quanto "con2" apontam para o mesmo objeto de conexão, tanto que fiz ao final do código um teste para comprovar isso. Ambos os ponteiros estão direcionados para a mesma posição de memória, a variável privada interna estática do Singleton.

Builder

O padrão Builder tem a intenção de separar a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações (Gamma et al., 1995). O padrão pode ser aplicado quando o algoritmo para a construção de um objeto deve ser independente das partes que o compõem e de como elas são montadas. O processo de construção deve permitir diferentes representações para o objeto que é construído. A aplicação deste padrão traz alguns benefícios, como permitir variar a representação interna de um produto, pois

uma interface permite ao construtor ocultar essa representação. O padrão melhora a modularidade pelo encapsulamento da forma como um objeto complexo é construído e representado. Em outras palavras, ele isola o código da construção.

Os participantes do padrão são os seguintes:

• Builder – especifica uma interface abstrata para a criação de partes de um objeto-produto;

• Concrete Builder – constrói e monta partes do produto pela implementação da interface de Builder; define e mantém a representação que cria; fornece uma interface para a recuperação do produto;

• Director – constrói um objeto usando a interface de Builder;

• Product – representa o objeto complexo em construção; Concrete Builder constrói a representação interna do produto e define o processo pelo qual ele é montado; inclui classes que definem as partes constituintes, inclusive as interfaces para a montagem das partes do resultado final.

O padrão Builder é muito parecido com um outro padrão chamado Template Method. Basicamente uma classe que precisa realizar uma série de tarefas não sabe como cada uma deve ser feita, mas sabe que é sempre na mesma ordem. Então, classes descendentes não precisam repetir a chamada a esses métodos na mesma ordem, basta apenas implementá-los, pois a ordem já está definida e orquestrada.

Recorremos mais uma vez a um exemplo da vida real para entender melhor a OO. Uma montadora de automóveis é capaz de produzir "n" modelos diferentes de veículos. Não importa o tipo de carro, se um modelo antigo ou novo, os passos para montar um veículo são sempre os mesmos, por exemplo, poderia se iniciar pela montagem do casco, depois pintura, portas, vidros, pneus e acabamento final interno. Se um novo modelo for lançado pela fábrica, a linha de montagem não é alterada, o que muda é o produto, mas não a sequência de passos (procedimentos). É exatamente isso que faz um Builder: ele permite definir a ordem de criação de "peças" de um objeto complexo, que não poderia ser criado de uma única vez.

A **Listagem 9** mostra um exemplo de um framework para construção de relatório e a **Figura 3** exibe sua representação gráfica.

Vamos começar observando a classe produto, a que define um relatório. Todos sabem que alguns relatórios possuem partes que quando construídas formam o objeto completo, por exemplo, no caso de um relatório, temos cabeçalho, grupos, corpo / detalhes, rodapé, sumários etc. A ordem de criação destas partes normalmente é sempre a mesma, não faz sentido um relatório ter um rodapé no topo da página. Os detalhes (details) são sempre a parte principal do relatório, que compõem a maior parte dos dados exibidos no mesmo. No entanto, digamos que classes clientes não soubessem como essa ordem deveria ser chamada para criação do relatório. Esse é o papel da classe seguinte na mesma unit, o Builder. Repare que ele tem os métodos de "build", um para cada parte (cabeçalho, corpo, rodapé). Esses métodos são virtuais, e são então sobrescritos com polimorfismo nas classes descendentes (os Concrete Builders), que são as classes que realmente constroem os relatórios, nesse caso, de clientes e produtos. Nesse ponto, um quarto personagem entra em cena, o gerador de relatórios. Ele contém um método constructRelatorio que sabe qual a ordem correta para criar um produto (relatório) a partir de um Builder. Nesse ponto, ele funciona de forma muito parecida a um Template Method.

A **Listagem 10** mostra como utilizar o framework para criação de relatórios complexos. Basicamente criamos dois relatórios, clientes e produtos, através dos seus respectivos Builders. O gerador garante que o cabeçalho vai ser criado primeiro, depois os detalhes, e finalmente o rodapé. Um procedimento no programa principal simula a exibição dos relatórios criados, varrendo as partes dos produtos complexos construídas com auxílio de Builders.

Listagem 8. Usando o Singleton

```
program Singleton;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  con1,con2: ConexaoBD;
begin
  con1 := ConexaoBD.Instance();
  con1.stringConexao := 'SQL Server';
  con1.Open();
  con2 := ConexaoBD.Instance();
  con2.Open();
  // comprovando que são a mesma instância
  // está referenciado a mesma posição de memória,
  // apontando para o mesmo objeto
  if (con1 = con2) then
    WriteLn('Apontam para a mesma instância');
  ReadLn;
end.
```

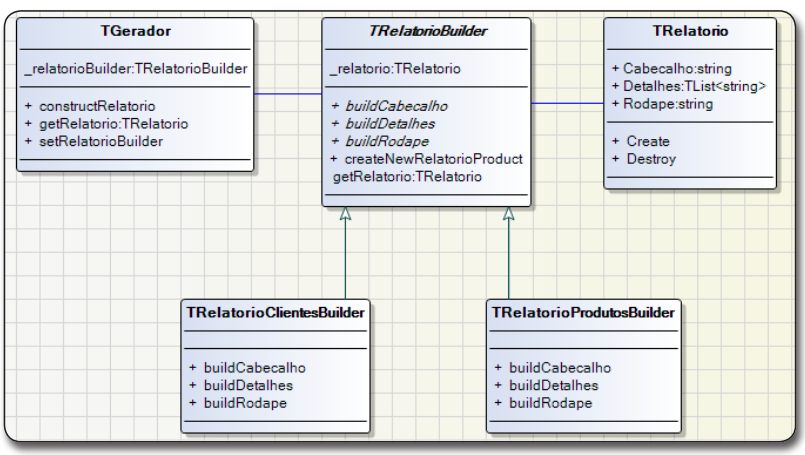

Figura 3. Estrutura de classes do exemplo usando Builder

Listagem 9. Builder

```
unit uFramework;

interface

uses
 Generics.Collections;

type
 TRelatorio = class
   public Cabecalho: string;
   public Detalhes: TList<string>;
   public Rodape: string;
   constructor Create();
   destructor Destroy(); override;
 end;

 // Builder (Abstrato)
 TRelatorioBuilder = class abstract
  protected _relatorio: TRelatorio;
  function getRelatorio(): TRelatorio;
  // processo de construção (abstrato)
  public procedure buildCabecalho(); virtual; abstract;
  public procedure buildDetalhes(); virtual; abstract;
  public procedure buildRodape(); virtual; abstract;
  public procedure createNewRelatorioProduct();
 end;

 // Builder (Concreto)
 TRelatorioClientesBuilder = class(TRelatorioBuilder)
  public procedure buildCabecalho(); override;
  public procedure buildDetalhes(); override;
  public procedure buildRodape(); override;
 end;

 // Builder (Concreto)
 TRelatorioProdutosBuilder = class(TRelatorioBuilder)
  public procedure buildCabecalho(); override;
  public procedure buildDetalhes(); override;
  public procedure buildRodape(); override;
 end;

 { Director - responsável pelo gerenciamento
  da sequência correta de criação do objeto
  Recebe um um Concret Builder como parâmetro e
  executa as operações necessárias sobre ele  }
 TGerador = class
  private _relatorioBuilder: TRelatorioBuilder;
  public procedure setRelatorioBuilder(relatorioBuilder: TRelatorioBuilder);
  public function getRelatorio(): TRelatorio;
  public procedure constructRelatorio();
 end;

implementation

{ RelatorioClientesBuilder }

procedure TRelatorioClientesBuilder.buildCabecalho();
begin
 _relatorio.Cabecalho := 'Relatório de Clientes';
end;

procedure TRelatorioClientesBuilder.buildDetalhes();
begin
 _relatorio.Detalhes.Add('Jose');
 _relatorio.Detalhes.Add('Pedro');
 _relatorio.Detalhes.Add('Joao');
end;

procedure TRelatorioClientesBuilder.buildRodape();
begin
 _relatorio.Rodape := 'Guinther Pauli';
end;

{ RelatorioProdutosBuilder }

procedure TRelatorioProdutosBuilder.buildCabecalho();
begin
 _relatorio.Cabecalho := 'Relatório de Produtos';
end;

procedure TRelatorioProdutosBuilder.buildDetalhes();
begin
 _relatorio.Detalhes.Add('iPhone');
 _relatorio.Detalhes.Add('iPad');
 _relatorio.Detalhes.Add('iPod');
end;

procedure TRelatorioProdutosBuilder.buildRodape();
begin
 _relatorio.Rodape := 'Guinther Pauli';
end;

{ RelatorioBuilder }

procedure TRelatorioBuilder.createNewRelatorioProduct();
begin
 _relatorio := TRelatorio.Create();
end;

function TRelatorioBuilder.getRelatorio(): TRelatorio;
begin
 result := _relatorio;
end;

{ Director }

procedure TGerador.constructRelatorio();
begin
 _relatorioBuilder.createNewRelatorioProduct();
 _relatorioBuilder.buildCabecalho();
 _relatorioBuilder.buildDetalhes();
 _relatorioBuilder.buildRodape();
end;

function TGerador.getRelatorio(): TRelatorio;
begin
 result := _relatorioBuilder.getRelatorio();
end;

procedure TGerador.setRelatorioBuilder(
 relatorioBuilder: TRelatorioBuilder);
begin
 self._relatorioBuilder := relatorioBuilder;
end;

{ Relatorio }

constructor TRelatorio.Create();
begin
 Detalhes := TList<string>.Create();
end;

destructor TRelatorio.Destroy();
begin
 Detalhes.Free();
 inherited;
end;

end.
```

Listagem 10. Usando o Builder

```
program Builder;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  uFramework in 'uFramework.pas';

var
  relClientes: TRelatorio;
  relProdutos: TRelatorio;
  relClientesBuilder: TRelatorioBuilder;
  relProdutosBuilder: TRelatorioBuilder;
  criador: TGerador;

procedure mostrarRelatorio(relatorio: TRelatorio);
var
  s: string;
begin
  Writeln(relatorio.Cabecalho);
  for s in relatorio.Detalhes do
    Writeln(s);
  Writeln(relatorio.Rodape);
end;

begin
  // cria o builder para relatório de clientes
  relClientesBuilder := TRelatorioClientesBuilder.Create();
  // cria director
  criador := TGerador.Create();
  // seta builder para relatório de clientes
  criador.setRelatorioBuilder(relClientesBuilder);
  // pede ao criador para montar relatório
  criador.constructRelatorio();
 // obtém o relatório de clientes criado
  relClientes := criador.getRelatorio();
  // mostra
  mostrarRelatorio(relClientes);
  // cria o builder para relatório de produtos
  relProdutosBuilder := TRelatorioProdutosBuilder.Create();
  // seta builder para relatório de produtos
  criador.setRelatorioBuilder(relProdutosBuilder);
  // pede ao criador para montar relatorio
  criador.constructRelatorio();
  // obtém o relatório de produtos criado
  relProdutos := criador.getRelatorio();
  // mostra
  mostrarRelatorio(relProdutos);
  Readln;
end.
```

Conclusão

Este artigo apresentou quatro padrões de projeto do grupo criacionais, destinados a reduzir o acoplamento entre classes através de abstrações e polimorfismo para criação de objetos. Comece tentando identificar no seu código onde poderia ser aplicado, por exemplo, um Abstract Factory, por exemplo, para isolar a camada de acesso a dados para torná-la mais fácil de ser substituída futuramente. Ou ainda, um Factory Method para reduzir aquele acoplamento direto entre sua interface de usuário e as classes de acesso a dados, ou ainda, classes de regras de negócio.

Autor

Guinther Pauli
http://www.facebook.com/guintherpauli
http://www.twitter.com/guintherpauli
http://br.linkedin.com/in/guintherpauli
http://www.gpauli.com
http://guintherpauli.blogspot.com.br
http://www.devmedia.com.br/guintherpauli

Arquiteto de Software certificado pela Microsoft, Consultor, Instrutor, Autor, Palestrante, Professor de Graduação e de Pós-Graduação, Colaborador Editorial das Revistas .NET Magazine Brasil e ClubeDelphi, Certificado oficial Microsoft MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET, C# e Arquitetura) - Certificado oficial Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Kylix. Most Valuable Professional (MVP) - Embarcadero Technologies. Entre os principais projetos que participou, foi arquiteto e desenvolvedor do sistema de doações on-line do projeto Criança Esperança da Rede Globo / Unesco. Beta Test de várias versões do Delphi, palestrante em 5 edições da Borland Conference, maior evento de desenvolvedores do país. Ministra treinamentos corporativos in-company, já tendo treinado profissionais de várias empresas dos estados do RS, SC, PR, SP, RJ, GO, TO, DF, MG, AM e PE. Autor de mais de 1100 videoaulas de ensino a distância pela DevMedia, publicou mais de 130 artigos, além do livro Delphi Programação para Banco de Dados e Web . Fez a cobertura de grandes eventos no exterior, como o TechEd da Microsoft e a BorCon da Borland americana. Experiência em Delphi de 18 anos e 13 anos em .NET / C#.

Links e Referências

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[2] M. Fowler, Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.

[3] J. Kerievsky. Refactoring to Patterns. Addison-Wesley, 2008.

GoF Patterns – Quick Review
http://www.mydeveloperconnection.com/html/gof_design_patterns.htm

Refactoring Home mantido pelo próprio Martin Fowler
http://www.refactoring.com

Refactorings in Alphabetical Order
http://www.refactoring.com/catalog/index.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Aplicando a RTTI para serializar informações

Utilizando a RTTI na prática

No Delphi, é possível tirar proveito do conceito de Reflexão Computacional através da Run-Time Type Information, ou como é mais conhecida, RTTI.

É através da RTTI que o próprio Delphi consegue em tempo de execução extrair e manipular informações de classes, objetos, componentes e etc. Um exemplo claro de uso da RTTI é a forma como o próprio IDE do Delphi lista as propriedades e eventos de um determinado componente no Object Inspector, que inclusive, foi o principal motivo que fez com que o Delphi passasse a ter suporte a reflexão computacional ainda em sua primeira versão. O mecanismo que torna possível a conversão dos dados armazenados em um arquivo .dfm em objetos, é a RTTI. Além destes, podemos citar ainda a forma como o framework DataSnap publica suas classes remotas e trafega informações entre o servidor e os clientes.

As versões mais recentes do DataSnap trafegam dados no formato texto em notação JSON. Apesar da possibilidade de camuflar esse tráfego em uma comunicação Delphi 2 Delphi (**BOX 2**), internamente, ocorre uma série de tratamentos utilizando a RTTI, o que viabiliza a comunicação entre as camadas envolvidas no processo. Ainda abordando o multicamadas, quando se está trabalhando com um servidor REST e clientes em outras linguagens, como JavaScript por exemplo, essa questão se torna evidente, devido ao fato do encapsulamento dos dados trafegados não serem camuflados.

Até a versão 2009 do Delphi, os recursos disponíveis da RTTI para o desenvolvedor eram extremamente

Resumo DevMan

Aplicando a RTTI para serializar informações:

A reflexão computacional (ou somente reflexão) é um tema muito antigo e consiste na capacidade de um sistema observar e interagir sobre o seu próprio comportamento. Através do conceito de reflexão, é possível obter informações sobre a estrutura e estado de um objeto de forma flexível, possibilitando o seu tratamento genérico.

É possível através de reflexão alterar o valor de um atributo ou até mesmo invocar um método de um determinado objeto sem que seja feita uma menção direta a sua instancia, como é feito tradicionalmente.

Em linguagens que não utilizam o paradigma reflexivo, no momento da compilação e geração do código de baixo nível (geralmente Assembly), as informações sobre a estrutura dos dados e objetos se perdem, porém, em uma linguagem com suporte a reflexão, essas informações são armazenadas e mantidas em tabelas de Metadados, fazendo com que seja possível obter informações sobre esta estrutura posteriormente, em tempo de execução.

A grande maioria das linguagens já possui suporte a reflexão e já fazem uso desta internamente, porém, o uso deste conceito por desenvolvedores é algo um pouco mais recente.

Outro exemplo a ser citado é a criação do Test Patterns xUnit, o padrão de projetos base para a criação do framework JUnit (**BOX 1**).

Em que situação o tema é útil:

Quando é necessário criar rotinas de serialização de dados, mapeamento de dados de forma dinâmica e criação de frameworks genéricos, de forma a reduzir a duplicidade de código e facilitar a manutenção do mesmo.

BOX 1. JUnit

O JUnit foi criado por Kent Beck e Erich Gamma, baseados no SUnit que é um framework de automação de testes unitários para Small Talk, criado por Kent Beck. O JUnit, assim como o SUnit, é utilizado para criação de testes unitários automatizados em Java que posteriormente serviu de base para a criação do DUnit, sua versão para Delphi. O padrão xUnit utiliza o conceito de reflexão computacional para generalizar, publicar e executar os métodos de testes unitários criados pelo desenvolvedor.

BOX 2. Delphi 2 Delphi

A forma mais popular de se trabalhar com o DataSnap é através de uma comunicação Delphi para Delphi, onde, no lado cliente, é utilizado o componente TSQLConnection do dbExpress para se criar uma comunicação com um servidor DataSnap. Por meio dessa conexão criada pelo componente TSQLConnection, é possível dentre outras coisas criar chamadas na aplicação cliente, que utilizam do Proxy para realizar as chamadas nos métodos remotos do servidor DataSnap. Para isso, basta utilizar-se da opção Generate DataSnap Client Classes no menu de contexto do componente TSQLConnection, quando este estiver com uma conexão DataSnap estabelecida.

limitados e de uso complexo. Porém, com o lançamento da versão 2010, a RTTI passou por significativas melhorias e seu uso passou a ser consideravelmente mais simples, fazendo com que o conceito viesse a se tornar gradativamente mais popular entre os desenvolvedores, ainda que a passos curtos.

Apesar dessa reformulação e da ainda pequena popularização da RTTI, o assunto continua extremamente teórico e com poucas aplicações práticas.

Inicialmente veremos um pouco sobre a base e conceitos da arquitetura da RTTI e, em seguida, aplicaremos alguns exemplos práticos, demonstrando como é simples obter todo o poder e versatilidade da RTTI no Delphi.

Principais estruturas utilizadas

O funcionamento da nova RTTI do Delphi está totalmente vinculada ao uso da unit RTTI.pas. Nessa unit estão localizados os tipos e principais métodos utilizados para tirar proveito de seus recursos. Dentre esses tipos, vale o destaque para as seguintes estruturas:

• TRttiContext - É o elemento chave do funcionamento da RTTI. O TRttiContext é utilizando para acessar as informações dos tipos públicos de uma aplicação e seus pacotes associados.

• TRttiType - É a classe base para todos os tipos que podem ser acessados pela RTTI. Através dela é possível expor métodos e atributos de um determinado tipo, capturado pelo TRttiContext.

• TValue - Consiste em uma estrutura de armazenamento de dados genérica, largamente utilizada nas rotinas da RTTI. Geralmente as instâncias de objetos, ou o conteúdo de variáveis e estruturas em RTTI são armazenados em estruturas do tipo TValue.

A partir dessas estruturas e outras auxiliares é possível tirar proveito de todo potencial da RTTI, como veremos a seguir.

O fluxo básico de funcionamento da RTTI no Delphi é o seguinte: Primeiramente é criado um contexto RTTI através do objeto TRttiContext, a seguir, são recuperadas informações sobre uma determinada estrutura, (seja uma classe ou um record) e estas são armazenadas em uma estrutura do tipo TRttiType. Essa extração de informações é feita também através de TRttiContext. A partir do da classe TRttiType é possível obter informações sobre campos (TRttiField), propriedades (TRttiProperty), métodos (TRttiMethod) e etc. No caso de campos e propriedades, é possível extrair informações sobre o tipo de dado que é armazenado, o valor que está armazenado neste campo em uma determinada instancia, o nome do campo, se ele possui algum Custom Attribute, a visibilidade deste campo dentro do encapsulamento (**BOX 3**), manipular o valor armazenado neste campo em uma determinada instancia e etc.

Quando o valor de um campo é manipulado, seja para leitura ou para escrita, isto é feito com auxílio da estrutura TValue. No caso de métodos, é possível extrair informações sobre o seu nome, seu tipo de retorno, sua visibilidade, seus parâmetros, seus Custom Attributes e também é possível invocar a sua execução a partir de uma instancia que possua o método em questão. Todos estes assuntos serão abordados a seguir.

Exemplo simples

Como já dito anteriormente, é possível manipular um objeto ou record através da RTTI, não só obter informações como também, os dados armazenados nessas estrutura. Sendo assim, faremos um exemplo bem simples de como obter a listagem de propriedades (property) de uma classe comum através da RTTI. A **Listagem 1** contém um exemplo.

BOX 3 . Encapsulamento

Em Delphi, existem 5 níveis de visibilidade para um membro de classe (seja ele um atributo ou um método), private, strict private, protected, public e published. Membros private são acessíveis somente dentro do escopo da própria classe e ou classes e métodos que façam parte da mesma unit. Para que um membro seja realmente privado para sua classe, se usa o nível de visibilidade strict private. O nível de visibilidade protected segue a mesma regra do nível private, porém, o membro passa a ser visível também para as classes que herdarem a classe que declarou este membro. Os níveis de visibilidade public e published, funcionam de maneira semelhante, deixando o membro com visibilidade externa total. A principal diferença entre os dois níveis fica por conta do uso de componentes. As propriedades de componentes que são exibidas no Object Inspector do Delphi em tempo de Design de projeto, são definidas como published, ou seja, sãs as propriedades publicadas. Na antiga RTTI, somente os membros de classe que estivessem declarados com o nível de visibilidade published eram possíveis de serem acessados.

Listagem 1. Classe de exemplo

```
01 ...
02 Type
03  TUsuario = class
04 strict private
05    FNome: string;
06    FSenha: string;
07    FLogin: string;
08    FUltimoAcesso: Double;
09  public
10    function EfetuarLogin():boolean;
11    function EfetuarLogoff():boolean;
12  published
13    property Nome : string read FNome write FNome;
14    property Login : string read FLogin write FLogin;
15    property Senha : string read FSenha write FSenha;
16    property UltimoAcesso : Double read FUltimoAcesso;
17  end;
```

Como se pode notar no código da **Listagem 1,** a classe TUsuario foi definida com 4 atributos dentro do escopo strict private (linhas 04 a 08). Além disso, foram definidos dois métodos públicos (linhas 10 e 11) e por último, a definição das 4 propriedades no escopo published (necessário para exibir no Object Inspector – linhas 13 a 16).

A partir da classe definida, faremos o método que lista que lista as propriedades de uma determinada classe. Para esse exemplo, será necessário um formulário com um componente TMemo (com o nome de mmInfo). A implementação do evento encontra-se na **Listagem 2**.

Listagem 2. Método para listar propriedades de uma classe

```
01 uses
02  RTTI, System.SysUtils, Classes;

03 procedure ListarPropriedadesDaClasse(AClasse: TClass; ASaida : TStrings);
04 var
05  _ctx : TRttiContext;
06  _typ : TRttiType;
07  _prp : TRttiProperty;
08 begin
09  _ctx := TRttiContext.Create;
10  try
11   _typ := _ctx.GetType(AClasse);
12   ASaida.Add(format('Propriedades da classe %s',[AClasse.ClassName]));
13   for _prp in _typ.GetDeclaredProperties do
14    ASaida.Add(_prp.Name);
15  finally
16   _ctx.Free;
17  end;
18 end;
```

Antes de executar o método definido na **Listagem 2**, vamos explorar o código. Na linha 05 foi definida uma variável do tipo TRttiContext (_ctx). Com ela, é possível recuperar as informações de uma determinada classe, o que é feito utilizando o método GetType na linha 11. Existem duas opções do método GetType (possíveis através de sobrecarga de métodos). Um deles recebe como parâmetro uma estrutura do tipo TClass, ou seja, a referência para uma classe. Já o segundo, recebe um ponteiro com as informações de uma determinada estrutura. Este pode ser obtido através do método TypeInfo. Geralmente, esta forma é utilizada quando se está trabalhando com records junto à RTTI.

A variável _typ do TRttiType (declarada na linha 06) torna possível recuperar as propriedades contidas em si mesmo, por meio da linha 11 (atribuídas por meio de GetType). Assim, utilizamos a instrução "for in" para percorrer as propriedades contidas no método GetDeclaredProperties. Aqui vale uma ressalva. Existem dois métodos distintos, sendo eles GetProperties e GetDeclared-Properties, onde o primeiro obtém qualquer propriedade que esteja no escopo da estrutura, já o segundo, recupera apenas as propriedades declaradas diretamente na estrutura, ou seja, o GetProperties recupera tanto as propriedades da estrutura, utilizando o TRttiContext, como também as propriedades herdadas na mesma.

Para utilizar o método ListarPropriedadesDaClasse utilizamos o código da **Listagem 3.** O resultado é demonstrado na própria listagem.

Note que, apesar da teoria ser complexa, na prática, o uso da RTTI é algo simples e objetivo. Agora faremos um exemplo onde serão recuperados os dados armazenados nesta classe. A partir dele já veremos o uso de TValue. Ainda usando a ideia do primeiro formulário, utilizaremos um TMemo para apresentar o resultado. O código encontra-se descrito na **Listagem 4**.

Basicamente, no método ListarDadosDaClasse da **Listagem 4** é feito o mesmo que o anterior (ListarPropriedadesDaClasse da **Listagem 2**), com a diferença que estamos recuperando a estrutura da classe a partir de um objeto (parâmetro de entrada AObj) e além disso, estamos obtendo o valor armazenado nas propriedades, através do método GetValue (linha 14).

O método GetValue pede como parâmetro a instancia na qual estamos querendo obter o valor, que encontra-se armazenado na propriedade em questão. Este método retorna uma estrutura do tipo TValue, que como parte de sua estrutura, possui o método ToString que converte qualquer valor armazenado em uma string. A execução e resultado respectivamente deste código encontram-se na **Listagem 5**.

Listagem 3. Resultado do método ListarPropriedadesDaClasse

```
01 ListarPropriedadesDaClasse(TUsuario,mmInfo.Lines);

02 //RESULTADO:
03 Propriedades da classe TUsuario
04 Nome
05 Login
06 Senha
07 UltimoAcesso
```

Listagem 4. Método para listar dados armazenados em uma classe

```
01 uses
02  RTTI, System.SysUtils, Classes;

03 procedure ListarDadosDaClasse(AObj: TObject; ASaida : TStrings);
04 var
05  _ctx : TRttiContext;
06  _typ : TRttiType;
07  _prp : TRttiProperty;
08 begin
09  _ctx := TRttiContext.Create;
10  try
11   _typ := _ctx.GetType(AObj.ClassType);
12   ASaida.Add(format('Dados na classe %s',[AObj.ClassName]));
13   for _prp in _typ.GetDeclaredProperties do
14    ASaida.Add(format('%s:
          %s',[_prp.Name,_prp.GetValue(AObj).ToString]));
15  finally
16   _ctx.Free;
17  end;
18 end;
```

Listagem 5. Execução do método ListarDadosDaClasse

```
01 procedure TfrmPrincipal.ExecutarListarDados;
02 var
03  oUsuario : TUsuario;
04 begin
05  oUsuario := TUsuario.Create;
06  try
07   oUsuario.Nome  := 'Diego';
08   oUsuario.Login := 'drgarcia1986';
09   oUsuario.Senha := 'DevMedia';
10   ListarDadosDaClasse(oUsuario,mmInfo.Lines);
11  finally
12   oUsuario.Free;
13  end;
14 end;

15 //RESULTADO:
16 Dados na classe TUsuario
17 Nome: Diego
18 Login: drgarcia1986
19 Senha: DevMedia
20 UltimoAcesso: 0
```

Observe na **Listagem 5** que o código é simples. Na linha 03 é declarada uma variável do tipo TUsuario, onde a seguir, na linha 05 é instanciada. Entre as linhas 07 e 09 os valores são atribuídos às propriedades. Na linha 10 o método criado anteriormente é chamado, passando como parâmetros a instância da classe e o componente Memo, liberando o objeto da memória em seguida, na linha 12. O resultado exibido no Memo será o mesmo ilustrado entre as linhas 16 a 20.

A partir de agora, veremos alguns exemplos práticos de rotinas utilizadas no cotidiano que podem ser melhoradas através do uso da RTTI.

Serialização de dados

Em um contexto computacional, serialização de dados consiste em converter um objeto para o formato texto (independente da notação) ou formato binário, para que assim seja possível utilizá-los em rotinas de persistência ou até mesmo transmissão de informações.

A ideia é que seja possível serializar um objeto em um local e, posteriormente, desserializá-lo a partir do mesmo local ou de outro, de forma que o resultado desse processamento faça com que o objeto esteja no mesmo estado de quando foi serializado. O funcionamento do fluxo do processo de serialização pode ser visto na **Figura 1.**

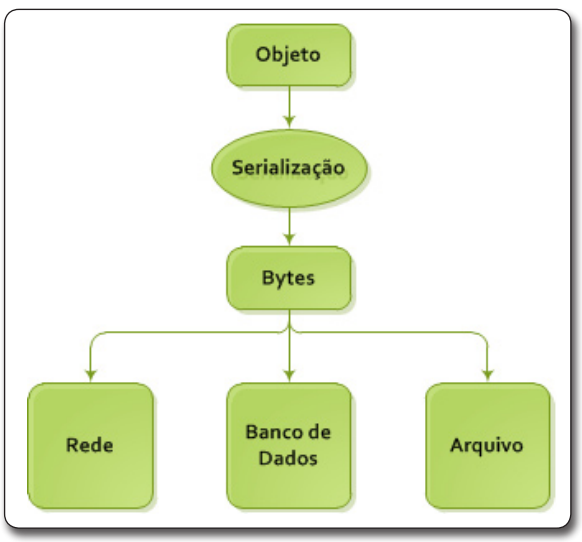

Figura 1. Fluxo simples de serialização de objetos

Um cenário onde muito comumente são utilizados recursos de serialização é a arquitetura de multicamadas (n-tier). Usando como exemplo a ferramenta mais popular para desenvolvimento de aplicações multicamadas do Delphi, o DataSnap, as informações são trafegadas como texto (hoje na notação JSON) e sofrem o processo de serialização/desserialização quando são entregues em suas respectivas pontas. Graças a esse recurso é possível trafegar qualquer objeto entre uma aplicação cliente e um servidor DataSnap. Além disso, o próprio DataSnap faz uso dos recursos da RTTI para serialização de objetos e posterior transmissão de dados.

Outro exemplo que está se tornando muito popular é a substituição do clássico arquivo INI por um arquivo de texto comum (contendo a serialização de um objeto na notação JSON), em rotinas de configurações locais de aplicativos.

Para demonstrar esse conceito de forma pratica, criaremos uma classe denominada TSerializeObject com o intuito de "tornar serializáveis" os objetos que forem herdados dela, de forma simples e genérica. Essa serialização será feita em notação JSON.

Visando simplificar os exemplos e tornar o conceito mais claro e fácil de assimilar, faremos o processo somente de 3 tipos básicos de dados, Unicode String, Integer e Real. A partir destes exemplos, a adaptação para outros tipos se torna simples, porém, para demonstrar o poder da RTTI, a serialização também funcionará com recursividade, ou seja, caso algum campo do objeto a ser serializado for outro objeto, este também participará da serialização. O código da classe TSerializeObject é demonstrado na **Listagem 6.**

Listagem 6. Interface da classe TSerializeObject

```
01 uses
02   DBXJSON, Classes, RTTI, TypInfo;

03 type
04   TSerializeObject = class abstract
05   public
06     function ToJSON():TJSONObject;
07     function ToFile(const AFilePath : string):boolean;
08     function FromJSON(AJSON : TJSONObject):boolean;
09     function FromFile(const AFilePath : string):boolean;
10   end;
```

No código da **Listagem 6** simplesmente foi realizada a declaração da classe TSerializeObject. Observe a declaração dos quatro métodos no escopo public. Iniciaremos implementando o método ToJSON. O objetivo desta função é serializar o objeto para a notação JSON e resultar um objeto do tipo TJSONObject. Sua implementação encontra-se na **Listagem 7.**

Analisando o método ToJSON da **Listagem 7**, temos algo bem semelhante ao que foi visto nas listagens anteriores. Observe que na linha 09 é utilizado GetType, atribuindo à variável _typ. A seguir, é utilizado o "for in" para percorrer os Fields contidos em GetDeclaredFields, dentro da estrutura de _typ. Da linha 13 em diante é utilizado o comando Case, validando o tipo do campo em si e realizando os Type Casts de acordo com o tipo do campo.

Nesta mesma listagem é possível notar o uso massivo da estrutura TValue nos fields por meio dos comandos GetValue. A partir de um TValue, é possível realizar o Type Cast para diversos tipos de dados, como foi feito anteriormente. É possível não só recuperar o valor, como também as informações sobre um campo. Estas informações são armazenadas em um TRttiField (o mesmo funciona com TRttiProperty por exemplo). Através desse recurso, foi possível determinar qual é o tipo de dado que é armazenado no campo e realizar a conversão corretamente.

Listagem 7. Método ToJSON da classe TSerializeObject

```
01 function TSerializeObject.ToJSON: TJSONObject;
02 var
03   _ctx : TRttiContext;
04   _typ : TRttiType;
05   _fld : TRttiField;
06 begin
07   _ctx := TRttiContext.Create;
08   try
09     _typ := _ctx.GetType(Self.ClassType);
10     Result := TJSONObject.Create;
11     for _fld in _typ.GetDeclaredFields do
12     begin
13       case _fld.FieldType.TypeKind of
14         tkUString :
15           Result.AddPair(_fld.Name,_fld.GetValue(Self).ToString);
16         tkInteger :
17           Result.AddPair(_fld.Name,
               TJSONNumber.Create(_fld.GetValue(Self).AsInteger));
18         tkFloat   :
19           Result.AddPair(_fld.Name,
               TJSONNumber.Create(_fld.GetValue(Self).AsExtended));
20         tkClass   :
21           Result.AddPair(_fld.Name,
             TSerializeObject(_fld.GetValue(Self).AsObject).ToJSON);
22       end;
23     end;
24   finally
25     _ctx.Free;
26   end;
27 end;
```

O próximo método a ser implementado (FromJSON) faz o processo inverso, ou seja, obter um JSON como parâmetro e desserializá-lo a partir disto. Sua implementação encontra-se na **Listagem 8**.

Assim como a função ToJSON, o método FromJSON da **Listagem 8** também faz uso massivo da estrutura TValue. Comentando passo a passo, obtemos novamente a referência da classe através de GetType (linha 11). A seguir, é percorrido o TJSONObject procurando por um TJSONPair, através da variável _Pair (linha 12). Na linha 14 é utilizado o método GetField para obter o objeto contido dentro do JSONPair, caso ele não esteja nulo (linha 15), é iniciado o processo de validação de tipos de acordo com o método TypeKind e a instrução Case.

Entre as linhas 17 a 26, os tipos são testados e o processo de desserialização é realizado, novamente utilizando o TValue e seus devidos TypeCasts. Se durante o processo tudo ocorrer corretamente o resultado da função será verdadeiro (linha 31). Na linha 33 finalmente, o objeto de contexto é destruído.

Antes de criarmos os métodos que fazem integração com arquivos (ToFile e FromFile) faremos um teste simples nos métodos que efetivamente realizam a serialização e a desserialização dos dados. Para isso, será realizada uma pequena alteração na classe TUsuario, já utilizada neste artigo em exemplos anteriores. As alterações podem ser vistas na **Listagem 9.**

A ideia na alteração da classe foi testar as possibilidades de serialização que criamos. Vale ressaltar que é possível a serialização de uma gama infinita de dados, porém, neste artigo foi proposto

Listagem 8. Método FromJSON da classe TSerializeObject

```
01 function TSerializeObject.FromJSON(AJSON: TJSONObject): boolean;
02 var
03   _ctx : TRttiContext;
04   _typ : TRttiType;
05   _fld : TRttiField;
06   _Pair : TJSONPair;
07   _Value : TValue;
08 begin
09   _ctx := TRttiContext.Create;
10   try
11     _typ := _ctx.GetType(Self.ClassType);
12     for _Pair in AJSON do
13     begin
14       _fld := _typ.GetField(_Pair.JsonString.Value);
15       if Assigned(_fld) then
16       begin
17         case _fld.FieldType.TypeKind of
18           tkUString :
19             _fld.SetValue(Self,
             TValue.From<string>(_Pair.JsonValue.Value));
20           tkInteger :
             _fld.SetValue(Self,
             TValue.From<Integer>(TJSONNumber(_Pair.JsonValue).AsInt));
21           tkFloat   :
             _fld.SetValue(Self,
             TValue.From<Double>(TJSONNumber(_Pair.JsonValue).AsDouble));
22           tkClass   :
23           begin
24             _Value := TValue.From<TSerializeObject>(
                   TSerializeObject(_fld.GetValue(Self).AsObject));
25           TSerializeObject(_Value.AsObject).FromJSON(
                   TJSONObject(_Pair.JsonValue));
26           _fld.SetValue(Self,_Value);
27         end;
28       end;
29     end;
30   end;
31   result := true;
32 finally
33   _ctx.Free;
34 end;
35 end;
```

apenas alguns tipos mais comuns visando a diminuição da complexidade do tema. Explorando o código da **Listagem 9** mais a fundo, note que primeiramente foi inserida uma nova classe (TDocumento), que descende de TSerializeObject e possui 2 propriedades públicas (linhas 03 a 10). Na classe TUsuario, fizemos com que ela, assim como a anterior, descendesse de TSerializeObject (linha 11), acrescentamos a propriedade Documento que é uma referência à classe TDocumento (linha 30). Nas linhas 22 e 23 foram adicionados o construtor e destrutor da classe respectivamente, que simplesmente inicializam o atributo FDocumento (construtor) e o destroem (destrutor) (linhas 33 a 40).

A seguir, na **Listagem 10** é demonstrado um exemplo de execução do método ToJSON visto anteriormente.

Como se pode notar, foram declaradas duas variáveis (linhas 02 e 03). Uma delas refere-se a classe TUsuario e a outra a um TJSONObject. Na linha 05 a variável JSON é inicializada e respectivamente, na linha 06, a classe TUsuario é criada. Entre as linhas 08 a 14 suas propriedades são alimentadas e na linha 15

Listagem 9. Alterações na classe TUsuario

```
01 interface

02 type
03  TDocumento = class(TSerializeObject)
04  strict private
05    FNumero: string;
06    FTipo: string;
07  public
08    property Tipo : string read FTipo write FTipo;
09    property Numero : string read FNumero write FNumero;
10  end;

11  TUsuario = class(TSerializeObject)
12  strict private
13    FNome: string;
14    FSenha: string;
15    FLogin: string;
16    FDocumento: TDocumento;
17    FIdade: integer;
18    FReferencia: real;
19  public
20    function EfetuarLogin():boolean;
21    function EfetuarLogoff():boolean;
22    constructor Create;
23    destructor Free;
24  published
25    property Nome : string read FNome write FNome;
26    property Login : string read FLogin write FLogin;
27    property Senha : string read FSenha write FSenha;
28    property Idade : integer read FIdade write FIdade;
29    property Referencia : real read FReferencia write FReferencia;
30    property Documento : TDocumento read FDocumento write FDocumento;
31  end;

32 implementation

33 constructor TUsuario.Create;
34 begin
35  Self.FDocumento := TDocumento.Create;
36 end;

37 destructor TUsuario.Free;
38 begin
39  Self.FDocumento.Free;
40 end;
```

Listagem 10. Exemplo de uso do método TSerializeObject.ToJSON

```
01 var
02  oUsuario : TUsuario;
03  oJSON : TJSONObject;
04 begin
05  oJSON := nil;
06  oUsuario := TUsuario.Create;
07  try
08    oUsuario.Nome  := 'Diego';
09    oUsuario.Login := 'drgarcia1986';
10    oUsuario.Senha := 'devmedia';
11    oUsuario.Idade := 26;
12    oUsuario.Referencia := 152;
13    oUsuario.Documento.Tipo := 'CPF';
14    oUsuario.Documento.Numero := '123.456.789-10';
15    oJSON := oUsuario.ToJSON;
17  finally
18    oUsuario.Free;
19    if Assigned(oJSON) then
20      oJSON.Free;
21  end;
22 end;
```

o método ToJSON retorna o JSON referente à própria classe, atribuindo à variável oJSON. Entre as linhas 18 a 20 as variáveis são destruídas e liberadas da memória. O resultado desta serialização provida pelo método ToJSON pode ser vista na **Listagem 11.**

Continuando a exploração da serialização, veremos agora o cenário inverso. Na **Listagem 12** conheceremos a maneira de uso do método FromJSON, onde, a partir de um TJSONObject, alimentaremos um objeto do tipo TUsuario.

Listagem 11. Resultado do método ToJSON

```
01 {
02  "FNome": "Diego",
03  "FSenha": "devmedia",
04  "FLogin": "drgarcia1986",
05  "FDocumento": {
06    "FNumero": "123.456.789-10",
07    "FTipo": "CPF"
08  },
09  "FIdade": 26,
10  "FReferencia": 152
11 }
```

Listagem 12. Exemplo de uso do método TSerializeObject.FromJSON

```
01 var
02  oUsuario : TUsuario;
03  oJSONPessoa : TJSONObject;
04  oJSONDocumento : TJSONObject;
05 begin
06  oUsuario := TUsuario.Create;
07  oJSONPessoa    := TJSONObject.Create;
08  oJSONDocumento := TJSONObject.Create;
09  try
10    oJSONDocumento.AddPair('FTipo','CPF');
11    oJSONDocumento.AddPair('FNumero','123.456.789-10');
12    oJSONPessoa.AddPair('FNome','Diego');
13    oJSONPessoa.AddPair('FLogin','drgarcia1986');
14    oJSONPessoa.AddPair('FSenha','devmedia');
15    oJSONPessoa.AddPair('FIdade',TJSONNumber.Create(26));
16    oJSONPessoa.AddPair('FReferencia',TJSONNumber.Create(152));
17    oJSONPessoa.AddPair('FDocumento',oJSONDocumento);
18    oUsuario.FromJSON(oJSONPessoa);
19  finally
20    oUsuario.Free;
21    oJSONPessoa.Free;
22  end;
23 end;
```

Assim como nos exemplos anteriores, temos a declaração das variáveis utilizadas entre as linhas 02 a 04. Entre as linhas 06 a 08 os objetos são criados. Note que foram declarados dois TJSONObject, isso porque devemos recuperar dados de duas classes distintas (TDocument e TUsuario). Entre as linhas 10 a 17 os valores são acrescentados utilizando o método AddPair, composto de "nome" e "valor". Por último na linha 18 o método FromJSON é executado, passando como parâmetro o objeto oJSONPessoa, destruindo as variáveis a seguir, nas linhas 20 e 21.

Finalizando o tema serialização, as **Listagens 13** e **14** respectivamente, mostram a implementação de "ToFile" e "FromFile", que nada mais são do que Wrappers para os métodos já descritos anteriormente.

Listagem 13. Método TSerializeObject.FromFile

```
01 function TSerializeObject.FromFile(const AFilePath: string): boolean;
02 var
03   oJSON : TJSONObject;
04   strlTemp : TStringList;
05 begin
06   oJSON := nil;
07   strlTemp := TStringList.Create;
08   try
09    strlTemp.LoadFromFile(AFilePath);
10    oJSON := TJSONObject(TJSONObject.ParseJSONValue(strlTemp.Text));
11    Result := Self.FromJSON(oJSON)
12   finally
13    strlTemp.Free;
14    oJSON.Free;
15   end;
16 end;
```

No código da **Listagem 13** podemos notar uma estratégia muito utilizada em Delphi para manipular arquivos texto através de um StringList. Nas linhas 03 e 04 são definidas as variáveis utilizadas. A linha 06 inicializa o objeto oJSON e, em seguida, na linha 07, o StringList é utilizado para importar o arquivo texto existente. Na linha 09 é utilizado o método LoadFromFile passando como parâmetro o caminho do arquivo em disco. A linha 10 como podemos notar realiza o Parse (conversão) do arquivo para um JSON e na sequência, este objeto é importado para dentro da classe através do método FromJSON, sendo retornado pela função.

Para finalizar, o método ToFile é ilustrado, conforme a **Listagem 14.**

Listagem 14. Método TSerializeObject.ToFile

```
01 function TSerializeObject.ToFile(const AFilePath: string): boolean;
02 var
03   oJSON : TJSONObject;
04   strlTemp : TStringList;
05 begin
06   oJSON := Self.ToJSON();
07   strlTemp := TStringList.Create;
08   try
09    strlTemp.Text := oJSON.ToString;
10    strlTemp.SaveToFile(AFilePath);
11    Result := true;
12   finally
13    strlTemp.Free;
14    oJSON.Free;
15   end;
16 end;
```

Essa listagem não requer grandes explanações já que possui seu código praticamente idêntico ao da **Listagem 13.** As únicas observações referentes aqui encontram-se na linha 09, na qual o StringList recebe o conteúdo do objeto JSON, realizando em seguida (linhas 10) a persistência em disco, através do método SaveToFile.

Atributos customizáveis (TCustomAttributes)

Para quem utiliza a RTTI, uma forte aliada com certeza é a classe TCustomAttribute. Os Custom Attributes, ou atributos customizáveis, são uma forma de estender as informações de uma determinada estrutura, seja ela uma classe, um atributo, um método ou um parâmetro, tanto para classes como para records.

Para exemplificar o uso de Custom Attribute, vamos criar uma regra que define que uma determinada propriedade de uma classe não pode ter um valor nulo, sendo zero ou menos, considerado nulo para números e um texto vazio considerado nulo para strings.

A **Listagem 15** demonstra a declaração deste Custom Attribute

Repare na listagem que a declaração do Custom Attribute nada mais é do que uma classe comum descendente da classe TCustomAttribute. Para vincular o nosso atributo customizável aos campos que iremos validar, basta fazer como o exemplo da **Listagem 16.**

Listagem 15. Atributo Customizável TNotNull

```
01 type
02   TNotNull = class(TCustomAttribute);
```

Listagem 16. Exemplo de uso do atributo TNotNull

```
01 TContato = class
02 private
03   FNome : string;
04   FIdade : integer;
05   FTelefone : String;
06   FEmail : string;
07   FSexo: byte;
08 public
09   [TNotNull]
10   property Nome : string read FNome write FNome;
11   [TNotNull]
12   property Idade : integer read FIdade write FIdade;
13   property Sexo : byte read FSexo write FSexo;
14   property Telefone : string read FTelefone write FTelefone;
15   [TNotNull]
16   property Email : string read FEmail write FEmail;
17 end;
```

Observando o código apresentado **16**, podemos notar que as definições são inseridas na classe por meio da tag "[TNotNull]" (propriedades entre as linhas 09 a 16). Desta forma, é possível criar um método genérico que valida os campos nulos em qualquer classe que possua o Custom Attributes TNotNull.

A **Listagem 17** faz uso das "marcações" definidas, apresentando o método ValidarCamposNulos, que através da RTTI e do uso de Custom Attributes, realiza esta validação em todas as propriedades de um objeto.

No método da **Listagem 17** foram definidas as variáveis entre as linhas 05 a 08. Observe que novamente foi utilizado o método GetType (linha 13), atribuindo o seu valor à variável _typ. A seguir, são percorridas as propriedades contidas dentro da variável, através de um laço de objetos "for in" (linha 14). A mágica acontece a partir da linha 16, onde o é utilizada a variável _ca para percorrer os Atributos de cada uma das propriedades. Note que na linha 18 o teste é realizado diretamente com a classe que criamos (**Listagem 16**) e que descendia de TCustomAttributes

(TNotNull). Da linha 20 em diante são validados os tipos para cada situação de nulo. Caso seja um tipo inteiro (linha 21), é testado se o valor da propriedade é inferior a zero (linha 23), atribuindo o nome da propriedade ao StringList ACamposNulos (recebido por parâmetro).

Listagem 17. Implementação do método ValidarCamposNulos

```
01 uses
02  RTTI, System.TypInfo, System.SysUtils;

03 function ValidarCamposNulos(AObjeto : TObject; ACamposNulos :
   TStringList):boolean;
04 var
05  _ctx : TRttiContext;
06  _ca  : TCustomAttribute;
07  _prp : TRttiProperty;
08  _typ : TRttiType;
09 begin
10  ACamposNulos.Clear;
11  _ctx := TRttiContext.Create;
12  try
13   _typ := _ctx.GetType(AObjeto.ClassType);
14   for _prp in _typ.GetDeclaredProperties do
15   begin
16    for _ca in _prp.GetAttributes do
17    begin
18     if _ca is TNotNull then
19     begin
20      case _prp.PropertyType.TypeKind of
21       tkInteger:
22       begin
23        if _prp.GetValue(AObjeto).AsInteger <= 0 then
24         ACamposNulos.Add(_prp.Name);
25       end;
26       tkUString:
27       begin
28        if _prp.GetValue(AObjeto).ToString = EmptyStr then
29         ACamposNulos.Add(_prp.Name);
30       end;
31      end;
32      break;
33     end;
34    end;
35   end;
36  finally
37   _ctx.Free;
38   Result := ACamposNulos.Count = 0;
39  end;
40 end;
```

Por outro lado, se o a propriedade é do tipo String, testamos a comparação com o método EmptyStr (linha 28) para saber se é nula e, em caso positivo, adicionamos seu conteúdo no StringList. Por fim, na linha 37 temos a destruição da variável e na linha 38 o retorno que será verdadeiro caso não haja nulos no StringList. O exemplo de uso do método ValidarCamposNulos pode ser visto no código da **Listagem 18**.

O código é bem semelhante ao que já vimos nas listagens anteriores. Observe que são criados os objetos oContato e slValidacao (linhas 05 e 06). As propriedades "Nome" e "Telefone" são preenchidas, enquanto as demais não, sendo assim, na linha 10 é efetuada a validação do preenchimento das propriedades, através do método ValidarCamposNulos, passando o objeto e o StringList como parâmetros. Caso haja nulos, o retorno baseado no conteúdo do será a mensagem definida na linha 11 e exibida na **Figura 2**.

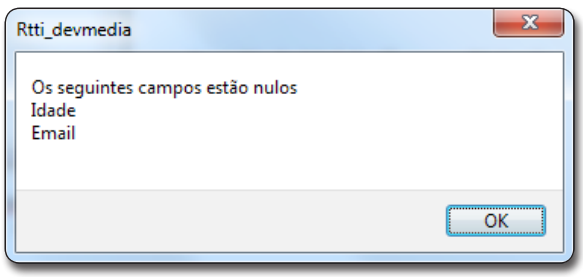

Figura 2. Resultado da validação com atributos customizáveis

Listagem 18. Exemplo de uso do método ValidarCamposNulos

```
01 var
02  oContato: TContato;
03  slValidacao: TStringList;
04 begin
05  slValidacao := TStringList.Create;
06  oContato := TContato.Create;
07  try
08   oContato.Nome := 'João';
09   oContato.Telefone := '55 11 98765-4321';
10   if not ValidarCamposNulos(oContato, slValidacao) then
11    ShowMessageFmt('Os seguintes campos estão nulos%s%s', [sLineBreak,
      slValidacao.Text]);
12  finally
13   oContato.Free;
14   slValidacao.Free;
15  end;
16 end;
```

É possível utilizar os Custom Attributes com parâmetros. Basta especificar os parâmetros necessários em seu construtor. Para exemplificar essa situação criaremos uma nova regra que irá definir o tamanho máximo de uma string, porém, esse tamanho não será fixo, devendo ser analisado caso a caso. **A Listagem 19** contempla a declaração do Custom Attribute TTamanhoMaximo.

A atribuição do Custom Attribute TTamanhoMaximo (**Listagem 19**) passa a ser um pouco diferente do que fizemos com o TNotNull, pois agora iremos definir nossa regra para cada situação. Este atributo possui algumas particularidades, tais como o construtor que receberá parâmetros (linha 06) e uma função de validação do tamanho (linha 07). Suas implementações encontram-se na mesma listagem, entre as linhas 10 e 17.

A **Listagem 20** demonstra um exemplo de como aplicaremos o Custom Attribute TTamanhoMaximo na própria classe TContato, utilizada anteriormente.

Note que, a atribuição do Custom Attribute no objeto é semelhante a declaração do seu construtor (linha 10), pois assim informarmos a ele quais são seus valores. Outra observação interessante é o fato de ser possível combinar Custom Attributes para os campos, como foi feito no campo Nome, que possui

Listagem 19. Atributo TTamanhoMaximo

```
01 type
02  TTamanhoMaximo = class(TCustomAttribute)
03  private
04   FTamanhoMaximo : integer;
05  public
06   constructor Create (ATamanhoMaximo : integer);
07   function TamanhoPermitido(ATamanho : integer):boolean;
08  end;

09 implementation

10 constructor TTamanhoMaximo.Create(ATamanhoMaximo: integer);
11 begin
12  Self.FTamanhoMaximo := ATamanhoMaximo;
13 end;

14 function TTamanhoMaximo.TamanhoPermitido(ATamanho: integer): boolean;
15 begin
16  Result := ATamanho <= Self.FTamanhoMaximo;
17 end;
```

Listagem 20. Aplicando o atributo TTamanhoMaximo na classe TContato

```
01 TContato = class
02  private
03   FNome : string;
04   FIdade : integer;
05   FTelefone : String;
06   FEmail : string;
07   FSexo: byte;
08  public
09   [TNotNull]
10   [TTamanhoMaximo(20)]
11   property Nome : string read FNome write FNome;
12   [TNotNull]
13   property Idade : integer read FIdade write FIdade;
14   property Sexo : byte read FSexo write FSexo;
15   property Telefone : string read FTelefone write FTelefone;
16   [TNotNull]
17   property Email : string read FEmail write FEmail;
18  end;
```

tanto o Custom Attribute TNotNull como o TTamanhoMaximo (linhas 09 e 10). Para complementar o exemplo anterior, vejamos como seria a função genérica que valida o Custom Attribute TTamanhoMaximo na **Listagem 21.**

Essa listagem é semelhante ao que já vimos, sendo assim, devemos atentar apenas as principais mudanças que encontram-se nas linhas 16 a 18. Repare que na linha 16 é validado se o Custom Attribute refere-se à TTamanhoMaximo. Na linha 18 é checado se o valor da propriedade encontra-se na regra de tamanho anteriormente definida, caso não esteja, novamente a propriedade é atribuída ao StringList recebido como parâmetro. O exemplo de uso deste método é demonstrado a seguir, na **Listagem 22.**

Assim como nos exemplos anteriores, são definidos os objetos TContato e um TStringList (linhas 02 e 03). Posteriormente ambos são criados (linhas 05 e 06), onde temos a propriedade "Nome" preenchida (linha 08) e a validação realizada logo na linha 09. Entre as linhas 12 e 13 tem-se a destruição dos objetos utilizados. A **Figura 3** ilustra o resultado em execução.

Listagem 21. Implementação no método ValidarTamanhoMaximo(

```
01 function ValidarTamanhoMaximo(AObjeto : TObject; ACamposInvalidos :
   TStringList):boolean;
02 var
03  _ctx : TRttiContext;
04  _ca  : TCustomAttribute;
05  _prp : TRttiProperty;
06  _typ : TRttiType;
07 begin
08  ACamposInvalidos.Clear;
09  _ctx := TRttiContext.Create;
10  try
11   _typ := _ctx.GetType(AObjeto.ClassType);
12   for _prp in _typ.GetProperties do
13   begin
14    for _ca in _prp.GetAttributes do
15    begin
16     if _ca is TTamanhoMaximo then
17     begin
18      if not TTamanhoMaximo(_ca).TamanhoPermitido(
          Length(_prp.GetValue(AObjeto).ToString)) then
19       ACamposInvalidos.Add(_prp.Name);
20      break;
21     end;
22    end;
23   end;
24  finally
25   _ctx.Free;
26   Result := ACamposInvalidos.Count = 0;
27  end;
28 end;
```

Listagem 22. Exemplo de uso do método ValidarTamanhoMaximo()

```
01 var
02  oContato: TContato;
03  slValidacao: TStringList;
04 begin
05  slValidacao := TStringList.Create;
06  oContato := TContato.Create;
07  try
08   oContato.Nome := 'Diego da Rocha Garcia';
09   if not ValidarTamanhoMaximo(oContato, slValidacao) then
10    ShowMessageFmt('Os seguintes campos excederam seu tamanho
      máximo%s%s', [sLineBreak, slValidacao.Text]);
11  finally
12   oContato.Free;
13   slValidacao.Free;
14  end;
15 end;
```

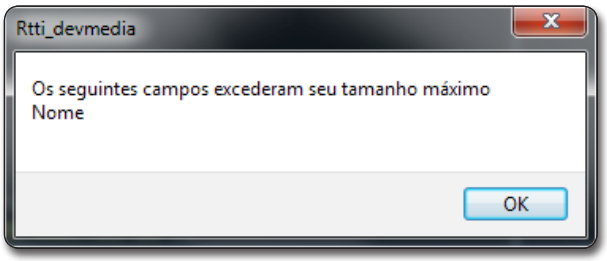

Figura 3. Resultado da validação de tamanho máximo

Conclusão

A reflexão computacional é um recurso extremamente poderoso, versátil e útil para o dia a dia de qualquer desenvolvedor. As últimas alterações e melhorias feitas na RTTI do Delphi fizeram com que o tema se tornasse mais simples e, consequentemente, mais popular para a comunidade de desenvolvedores pascal pelo mundo.

Neste artigo foram apresentadas introduções aos conceitos de reflexão computacional e serialização de dados, além de exemplos simples e didáticos que podem ser adaptados para rotinas genéricas que poupem um grande número de linhas de código, tornando-o mais simples e reaproveitável. Tudo através da RTTI, o framework de reflexão computacional do Delphi.

Apesar do tema ainda ser considerado algo muito complexo por alguns desenvolvedores, na prática, o uso da tecnologia é algo simples e de fácil aprendizado.

Autor

Diego Garcia
drgarcia1986@gmail.com
http://drgarcia1986.wordpress.com
@drgarcia1986

Analista/Desenvolvedor Delphi desde 2008, bacharel em Ciência da Computação e entusiasta de metodologias ágeis e engenharia de software. Atualmente escreve artigos sobre Delphi, Object Pascal e Firebird em seu blog e em publicações especializadas. e em publicações especializadas.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Envio de e-mails com boas práticas

Conheça três eficientes soluções para envio de e-mails

Com a onda da Web cada vez mais forte, as soluções de software começam a naturalmente disponibilizar recursos que se integram à Web, adotando serviços na arquitetura Cloud Computing, por exemplo, (**BOX 1**), integrações com redes sociais como Facebook e Twitter por meio do APIs (**BOX 2)** entre tantos outros recursos a serem mencionados. O maior exemplo que ilustra o atual cenário da importância da Web é algo na qual atualmente podemos visualizar em praticamente todos os sistemas de segmento comercial: interações com os serviços governamentais. Conhecimentos de Transporte eletrônicos (CT-e), Notas Fiscais eletrônicas (NF-e) e SPED são apenas alguns dos serviços mais utilizados para a interação entre o software e o governo, que passa cada dia mais a ser mais utilizado e com a exigência de novos segmentos.

BOX 1. Cloud Computing

Cloud Computing ou computação na nuvem é uma forma que prioriza a computação de forma 100% na Web. Isto permite com que as informações fiquem em ambientes externos, com inúmeros recursos que garantem a velocidade, segurança, escalabilidade, distribuição geográfica entre várias outras vantagens. Várias empresas fornecem serviços desta forma, entre as mais conhecidas podemos destacar a Amazon, Google e Microsoft.

BOX 2. API

API é o acrônimo para Application Programming Interface ou Interface de Programação de Aplicativos em uma tradução literal. É dada como uma forma de integração entre tecnologias heterogêneas, fazendo com que aplicações escritas nas mais diferentes linguagens, possam trocar informações entre si, independentes de ambiente.

Citando a nota fiscal eletrônica que é algo mais trivial, podemos notar que ao efetuar uma compra em um e-commerce (cenário Web), com a aprovação do pagamento, a nota fiscal eletrônica é transmitida ao site da receita federal e o documento DANFE (formato impresso da nota) é automaticamente expedido para o e-mail do consumidor (às vezes anexado com o arquivo XML, que comprova de fato a transmissão).

De fato este é apenas um exemplo comum da utilização da Web como forma de integração/comunicação entre serviços e softwares. O envio de e-mails é uma tarefa que passa despercebida em nosso cotidiano, porém, utilizando este recurso a nosso favor, temos uma poderosa ferramenta que com um pouco de criatividade, pode agregar muitas funcionalidades ao seu sistema. Softwares conhecidos, como é o caso do Final Builder (visto em artigos anteriores - **ver BOX 3**), permitem a configuração de envio de e-mails automaticamente, fazendo com que a equipe receba notificações sobre os processos realizados pela própria ferramenta (como término de uma compilação bem sucedida, erros, entre outros). Indo mais além, poderíamos facilmente adaptar este cenário a nossas aplicações, de modo com que em caso de um erro, fosse enviado um e-mail para o suporte com detalhes sobre o ocorrido, anexando à tela e inclusive efetuando respostas automáticas

Resumo DevMan

Envio de e-mails com boas práticas:

Em um cenário atual, praticamente todas as aplicações voltadas para área comercial fazem uso de envio de e-mails. Isto porque inúmeros são os processos que o sistema deve estar apto a notificar que uma determinada ação foi realizada com êxito. Que tal conhecer três soluções distintas para envio de e-mails fazendo uso de boas práticas de desenvolvimento? É justamente o que este artigo propõe.

Em que situação o tema é útil:

O tema é útil em praticamente qualquer aplicação que faça uso de envios de notificação ou arquivos de maneira automatizada, dispensando o uso de clientes de e-mail. Entre os mais comuns, podemos destacar o caso de vendas de produtos online, na qual o e-mail da nota fiscal é enviado para o consumidor final.

sobre o recebimento do erro, tornando este processo mais ágil, ou mesmo demonstrando o status de uma solicitação durante o seu processo de correção/desenvolvimento.

BOX 3. Final Builder

O Final Builder é uma ferramenta de melhoria de processos na qual tem por principal objetivo automatizar as funções de Build de projetos. Através dele, é possível definir regras específicas de compilação em caso de falhas ou acertos, compilação agendada, integrações com soluções de versionamento como o SVN ou StarTeam, compactação de arquivos, envio de e-mails automáticos definidos por regras, dentre diversas outras finalidades. Apesar de já conhecida, foi "incorporada" ao Delphi a partir da versão XE, na qual veio recheada de outras consagradas ferramentas de desenvolvimento.

Pois bem, todo este processo pode ser realizado fazendo uso de boas práticas de desenvolvimento, através de classes que separem as funcionalidades de envio de e-mail, possibilitando a reutilização e adaptação deste recurso para várias finalidades e nas mais variadas situações. Sendo assim, este artigo tem por finalidade adotar as boas práticas de desenvolvimento de software para demonstrar três diferentes soluções viáveis para o envio de e-mail, fazendo uso de seus principais recursos, além de demonstrar um comparativo das mesmas.

Conhecendo as soluções de e-mail

Para o desenvolvimento do nosso exemplo, utilizaremos três soluções conforme já mencionado. Sendo assim, iremos conhecer um pouco mais a respeito delas, suas vantagens, desvantagens e a seguir, elaborar o exemplo prático que no decorrer do artigo será melhorado.

1 – Indy Project

O Indy Project ou apenas Indy é um velho conhecido dos desenvolvedores Delphi. Trata-se de uma biblioteca de componentes Open-Source que a princípio surgiu como opção para o suporte de aplicações Web, permitindo a integração com e-mail através de serviços de smtp e pop, ftp entre muitos outros. Acompanha o Delphi desde muito tempo e é um dos pilares da arquitetura do framework DataSnap.

Prós:

- Open-Source;
- Solução nativa no Delphi, dispensando a instalação de componentes;
- Fácil integração e manuseio;
- Grande quantidade de material disponível na Web.

Contras:

- O projeto possui uma quantidade imensa de versões das DLLs, como a ssleay32.dll e libeay32.dll, utilizadas para suporte ao SSL (**BOX 4**), gerando muitas vezes problemas de incompatibilidade;
- Exige a distribuição das DLLs.

2 - Synapse

Assim como o Indy, o Synapse trata-se de uma biblioteca Open-Source para a mesma finalidade. Ao contrário do Indy ela não utiliza de componentes e sim Units para a integração, no entanto, ainda faz uso das mesmas DLLs ssleay32.dll e libeay32.dll. É muito conhecida por estar presente em uma das principais suíte de automação para Delphi, o projeto ACBr.

Prós:

- Open-Source;
- Por ser um conjunto de units, dispensa qualquer instalação, bastando apenas que a unit esteja presente no projeto ou na Library do Delphi;

Contras:

- Embora não seja difícil utilizá-lo, ainda assim não é simples quanto as outras soluções;
- Há pouco material na Web;
- Exige a distribuição das dlls como o Indy.

BOX 4. SSL

SSL é o acrônimo para Secure Sockets Layer. Basicamente são protocolos criptografados que são exigidos para conexões seguras com serviços SMTP ou navegação HTTP. Atualmente praticamente todos os serviços de Web requerem conexões seguras como é o caso do Gmail e Twitter por exemplo.

3 – Clever Internet Suite

A Clever Internet Suite da Clever Components (sessão **Links** ao final do artigo), como o próprio nome sugere, é uma suíte para total integração à Web. Possui suporte a diversos recursos por meio de componentes. Entre eles podemos destacar componentes para consumo de APIs como Facebook, Twitter, Google entre várias outras.

Prós:

- Simples instalação;
- Possui grande diversidade de componentes que consomem ou interagem com serviços da Web;
- Simples de usar;
- Não utiliza DLLs, encapsulando tudo dentro dos próprios componentes.

Contras:

- Embora os componentes sejam excelentes, são pagos e possuem um preço relativamente elevado;
- O site possui alguns exemplos com os principais componentes, no entanto, ainda assim o material a respeito da suíte é escasso na Web.

Criando a aplicação de exemplo

Iniciaremos criando uma aplicação comum VCL Forms, lembrando que o objetivo não será criar uma solução completa de e-mails, mas sim apresentar soluções para a implementação utilizando boas práticas. Sendo assim, inicie um projeto novo em File>New>VCL Forms Application – Delphi. Nome o formulário como frmPrincipal e salve o projeto, nomeando a unit como uPrincipal.pas e o projeto como DevMail.dproj.

No formulário principal, insira no topo um Panel de alinhamento alClient e, dentro dele, quatro edits renomeados respecti-

vamente como edtSMTP, edtPorta, edtUsuario e edtSenha. Logo abaixo, adicione mais três edits (edtDe, edtPara e edtAssunto), um Memo (memTexto), 1 RadioGroup (rgEmail) e por último, um BitBtn (btnEnviar). O Layout da sua tela deve estar semelhante ao da **Figura 1**.

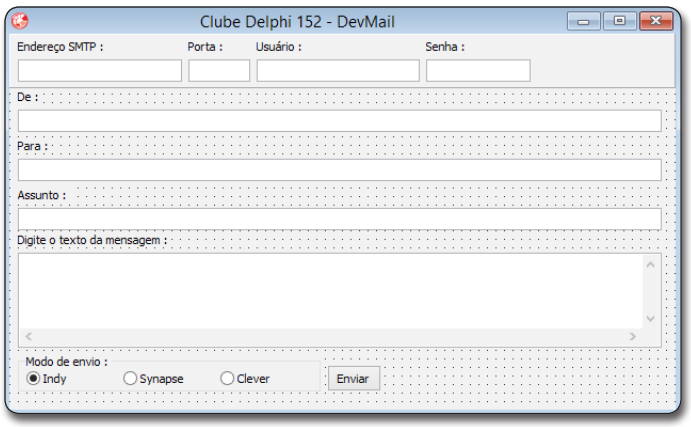

Figura 1. Layout da aplicação de envio de e-mails

Nota:

A implementação de anexo só será realizada nas classes do Indy, porém, pode ser facilmente adaptada para as outras classes a partir do mesmo conceito.

Após ter definido o layout, faremos a implementação da primeira classe. Selecione a opção File>New>Unit – Delphi e salve-a com o nome de uEmailIndy.pas. A **Listagem 1** ilustra o protótipo da nossa classe uEmailIndy após definida.

Como pode-se notar no código da **Listagem 1**, entre as linhas 03 a 08 temos as units necessárias para o uso da biblioteca Indy. A seguir, entre as linhas 12 a 25 temos os atributos privados da classe, na qual veremos mais adiante. Nas linhas 26 a 45 os métodos de acesso de cada propriedade (get e set) e por último, entre as linhas 47 a 61, estão as propriedades em si os métodos (Anexar e Enviar), construtor e destrutor da classe respectivamente.

Com isso temos a definição da primeira classe do projeto. A implementação dos métodos get e set respectivamente devem ser realizados para cada uma das propriedades, considerando o atributo correspondente. A **Listagem 2** ilustra um exemplo utilizando a propriedade "Porta".

Nota:

O código completo deste exemplo estará disponível para download no portal da DevMedia.

Seguindo as implementações necessárias vistas de acordo com a **Listagem 1**, o primeiro passo é criar os objetos que serão utilizados mais adiante no método Enviar, sendo assim, a **Listagem 3** ilustra o construtor da classe TEmailIndy

Listagem 1. Definiçao da classe uEmailIndy

```
01 unit uEmailIndy;

02 interface

03 uses
04  IdTCPConnection, IdTCPClient, IdExplicitTLSClientServerBase,
05  IdMessageClient, IdSMTPBase, IdSMTP, IdMessage, IdComponent,
06  IdIOHandler, IdIOHandlerSocket, IdIOHandlerStack, IdSSL, IdSSLOpenSSL, 07
IdBaseComponent, IdAntiFreezeBase, IdAttachmentFile, Vcl.IdAntiFreeze, 08   System.
Classes, System.SysUtils, Forms, Windows;

09 type
10  TEmailIndy = class
11  private
12    FIdAntiFreeze : TIdAntiFreeze;
13    FIdMessage : TIdMessage;
14    FIdSSLIOHandlerSocketOpenSSL : TIdSSLIOHandlerSocketOpenSSL;
15    FIdSMTP : TIDSMTP;
16    FDestinatarioEmail: String;
17    FRemeteNteNome: String;
18    FPorta: Integer;
19    FRemeteNteEmail: String;
20    FAssunto: String;
21    FSenha : String;
22    FDestinatarioNome: String;
23    FMensagem : String;
24    FHost : String;
25    FUsuario : String;
26    procedure SetAssunto(const Value: String);
27    procedure SetDestinatarioEmail(const Value: String);
28    procedure SetDestinatarioNome(const Value: String);
29    procedure SetRemeteNteEmail(const Value: String);
30    procedure SetRemeteNteNome(const Value: String);
31    procedure SetUsuario(const Value: String);
32    procedure SetSenha(const Value : String);
33    procedure SetPorta(const Value : Integer);
34    procedure setMensagem(const Value : String);
35    procedure setHost(const Value : String);
36    function getAssunto : String;
37    function getDestinatarioNome : String;
38    function getDestinatarioEmail : String;
39    function getRemeteNteEmail : String;
40    function getRemeteNteNome : String;
41    function getMensagem : String;
42    function getSenha : String;
43    function getHost : String;
44    function getUsuario : String;
45    function getPorta : Integer;
46  public
47    property RemeteNteEmail : String read getRemeteNteEmail write
      SetRemeteNteEmail;
48    property DestinatarioEmail : String read getDestinatarioEmail write
      SetDestinatarioEmail;
49    property RemeteNteNome : String read getRemeteNteNome write
      SetRemeteNteNome;
50    property DestinatarioNome : String read getDestinatarioNome write
      SetDestinatarioNome;
51    property Assunto : String read getAssunto write SetAssunto;
52    property Usuario : String read getUsuario write SetUsuario;
53    property Mensagem : String read getMensagem write setMensagem;
54    property Senha : String read getSenha write SetSenha;
55    property Porta : Integer read getPorta write SetPorta;
56    property Host : string read getHost write setHost;
57    procedure Anexar(Arquivo : String); overload;
58    procedure Anexar(Arquivo : TStrings); overload;
59    procedure Enviar(SSL : Boolean = True); overload;
60    constructor Create(AOwner : TComponent);
61    destructor Destroy;
62  end;
```

Listagem 2. Exemplo de implementação dos métodos Get e Set da propriedade Porta

```
01 //propriedade
02 property Porta : Integer read getPorta write SetPorta;

03 //método Set
04 procedure TEmailIndy.SetPorta(const Value: Integer);
05 begin
06  FPorta := Value;
07 end;

08 //método Get
09 function TEmailIndy.getPorta: Integer;
10 begin
11  Result := FPorta;
12 end;
```

Listagem 3. Construtor da classe TEmailIndy

```
01 constructor TEmailIndy.Create(AOwner : TComponent);
02 begin
03  inherited Create;
04  FIdAntiFreeze := TIdAntiFreeze.Create(AOwner);
05  FIdMessage := TIdMessage.Create(AOwner);
06  FIdSSLIOHandlerSocketOpenSSL :=
    TIdSSLIOHandlerSocketOpenSSL.Create(AOwner);
07  FIdSMTP := TIdSMTP.Create(AOwner);
08 end;
```

Embora muitas vezes não implementado de forma explícita, toda classe possui um construtor. Quando não está explícito o Construtor padrão da classe base é automaticamente chamado. Neste caso, o construtor da classe foi reescrito. Na linha 03 tem-se a chamada do construtor padrão da classe, a seguir o objeto FIdAntiFreeze é criado (linha 04). Sua finalidade como o próprio nome sugere, é evitar que a aplicação congele enquanto o envio de e-mail está sendo realizado. A linha 05 cria um objeto TIdMessage, que trata-se do e-mail em si. A classe TIdSSLIOHandlerSocketOpenSSL (linha 06) é utilizada para autenticação de forma segura no serviço de e-mail e por último, na linha 07, o objeto FIdSMTP é criado. Este será o responsável pela conexão com o servidor SMTP e pelo envio da mensagem em si.

Assim como todas as classes possuem explicitamente ou não o construtor, é possível implementar os destrutores. Os destrutores especificam o que deve ser feito quando a classe é destruída. Neste caso, ele simplesmente faz com que os objetos criados sejam automaticamente destruídos, chamando o destrutor original da classe, conforme pode ser notado na **Listagem 4.**

O segredo para o envio de e-mail em nossas classes concentra-se basicamente no método Enviar. Este conterá as particularidades de acordo com cada uma das soluções escolhidas. A seguir, na **Listagem 5**, é possível ver a implementação do método Enviar para a classe TEmailIndy.

Considerando a **Listagem 5**, pode-se notar uma validação de acordo com o parâmetro de entrada SSL do método enviar (linha 05). Sendo ele verdadeiro, o componente FIdSMTP receberá as configurações de autenticação de forma segura (ver **BOX 5**) através do objeto FIdSSLIOHandlerSocketOpenSSL, caso contrário, a propriedade IOHandler (linha 14) receberá nil, o que significa que não há requerimento de uma conexão segura. Entre as inhas 15 e 21 são realizadas as configurações para a conexão com o servidor de e-mail (servidor SMTP). Note aqui que algumas delas já fazem uso das propriedades anteriormente

Listagem 4. Destrutor da classe de E-mails

```
01 destructor TEmailIndy.Destroy;
02 begin
03  FreeAndNil(FIdAntiFreeze);
04  FreeAndNil(FIdMessage);
05  FreeAndNil(FIdSSLIOHandlerSocketOpenSSL);
06  FreeAndNil(FIdSMTP);
07  inherited Destroy;
08 end;
```

Listagem 5. Método Enviar da classe TEmailIndy

```
01 procedure TEmailIndy.Enviar(SSL : Boolean);
02 begin
03  with FIdSMTP do
04  begin
05   if SSL then
06   begin
07    IOHandler := FIdSSLIOHandlerSocketOpenSSL;
08    FIdSSLIOHandlerSocketOpenSSL.SSLOptions.Method := sslvSSLv3;
09    FIdSSLIOHandlerSocketOpenSSL.SSLOptions.Mode  := sslmClient;
10    FIdSSLIOHandlerSocketOpenSSL.SSLOptions.VerifyMode := [];
11    FIdSSLIOHandlerSocketOpenSSL.SSLOptions.VerifyDepth := 0;
12   end
13   else
14    IOHandler := nil;
15   UseTLS     := utUseExplicitTLS;
16   AuthType   := satDefault;
17   Host       := Self.Host;
18   Username   := Usuario;
19   Password   := Senha;
20   Port       := Porta;
21   ConnectTimeout := 60000;
22  end;
23  with FIdMessage do
24  begin
25   From.Name       := RemetenteNome;
26   From.Address    := RemetenteEmail;
27   From.DisplayName  := RemetenteNome;
28   Subject             := Assunto;
29   Sender.Address      := DestinatarioEmail;
30   Sender.Name         := DestinatarioNome;
31   Recipients.EMailAddresses := DestinatarioEmail;
32   Body.Text           := Mensagem;
33   Priority := mpNormal;
34  end;
35  try
36   try
37    FIdSMTP.ReadTimeout := 60000;
38    FIdSMTP.Connect;
39    FIdSMTP.Authenticate;
40    if FIdSMTP.Connected then
41     FIdSMTP.Send(FIdMessage)
42    Application.messageBox('E-mail enviado com sucesso!',
      'Confirmação!', MB_ICONINFORMATION + MB_OK);
43   except
44    on e: exception do
45     raise Exception.Create('Erro detectado: ' + e.Message);
46   end;
47  finally
48   FIdSMTP.Disconnect;
49  end;
50 end;
```

definidas, como é o caso de Host e UserName. Vale ainda salientar que as propriedades UseTLS e AuthType são particulares de acordo com cada serviço de e-mail (Hotmail, Gmail, Yahoo e etc). Já a propriedade ConnectionTimeout da linha 21, específica o tempo máximo permitido de conexão com o servidor. Entre as linhas 23 a 33 temos as configurações do componente de mensagem (FIdMessage) que conforme mencionado anteriormente, compõem o e-mail em si. As subpropriedades "From" (linhas 25 a 27) referem-se ao remetente, "Sender" tratam o destinatário e "Recipients" seriam utilizadas para o envio de mais de um destinatário, separados pelo caractere especial ";". Existem outras propriedades para este componente na qual não foram abordadas, como é o caso dos nomes de exibição do remetente e destinatário, porém, não são de uso obrigatório, podendo ser configuradas facilmente conforme a necessidade. A propriedade "Body" refere-se ao texto da mensagem que será enviada. Por último, entre as linhas 37 a 48 o envio é realizado de fato, usando o componente FIdSMTP. Na linha 37 é especificado o tempo permitido no processo. Caso este tempo seja excedido, será automaticamente gerada uma exceção do tipo "Connection Timeout". A seguir a conexão é realizada, o componente realiza a autenticação através das configurações de usuário, senha, porta e host. Caso este processo seja realizado com êxito, o servidor estará ativo e a mensagem será enviada através do método Send (linha 41). A seguir uma mensagem de êxito é dada ao usuário, encerrando a conexão finalmente, na linha 48.

Nota

Os principais processos que diferenciam as classes entre si (além das units) são os métodos construtores e Enviar. Por estender muito o artigo, as demais classes (TEmailSynalist e TEmailClever) não terão seus códigos-fonte explorados a fundo, no entanto, o código destas classes estará disponível juntamente com o exemplo no portal DevMedia.

Partindo do princípio de que as classes já estão criadas, voltaremos à atenção ao formulário principal, criando os devidos objetos e fazendo seu respectivo uso, porém, de modo convencional. O objetivo é de fato ilustrar como as boas práticas de desenvolvimento podem lhe render um código mais limpo e auxiliar na manutenção de um Software. Sendo assim, o primeiro passo é adicionar as units uEmailIndy, uEmailClever e uEmailSynalist ao uses de Interface do formulário principal e, em seguida, declarar os respectivos objetos no escopo private, conforme ilustra a **Listagem 6**.

Declarados os objetos, faremos a criação dos mesmos. É uma boa prática escrever métodos para inicialização das classes, variáveis de classe e concentração do comportamento inicial da tela, deste modo, foi criado método Initialize, que tem justamente a finalidade de inicializar as variáveis anteriormente declaradas. O código encontra-se na **Listagem 7.**

O próximo passo é chamar o método Initialize para que as classes sejam de construídas antes do uso. Para isso utilizaremos o evento OnCreate do formulário, conforme a **Listagem 8** sugere.

Listagem 6. Declaração das variáveis referentes as classes

```
01 ...
02  private
03    _EmailIndy   : TEmailIndy;
04    _EmailSynapse : TEmailSynalist;
05    _EmailClever  : TEmailClever;
```

Listagem 7. Método Initialize

```
01 procedure TfrmPrincipal.Initialize;
02 begin
03   _EmailIndy   := TEmailIndy.Create(Self);
04   _EmailSynapse := TEmailSynalist.Create(Self);
05   _EmailClever  := TEmailClever.Create(Self);
06 end;
```

Listagem 8. Evento OnCreate do formulário principal

```
01 procedure TfrmPrincipal.FormCreate(Sender: TObject);
02 begin
03   Initialize;
04 end;
```

Vale fazer uma ressalva de que o evento OnCreate poderia conter a implementação direta das classes no lugar do método Initialize, porém, utilizando o evento Initialize, o código fica centralizado, podendo facilmente ser portado ou estendido. Para finalizar esta primeira parte, resta a implementação do evento OnClick do botão de envio, que encontra-se implementada na **Listagem 9.**

Observe que a listagem, embora seja longa, é repetitiva. Isto foi realizado propositalmente, de modo a tornar mais óbvia a visualização das vantagens de se utilizar boas práticas de desenvolvimento. Observe que o RadioGroup (rgEmail) valida qual o tipo de envio de e-mail a ser realizado. Para a posição 0, será utilizada a classe do Indy para o envio do e-mail, efetuando o preenchimento das propriedades em tela. O mesmo é realizado no caso do ItemIndex 1 e 2, seguindo respectivamente as soluções do Synapse e Clever Components.

Ao final destas etapas a solução está pronta e enviando e-mails. Nas **Figuras 2** e **3** respectivamente, é possível visualizar o processo de envio e recebimento dos e-mails. A seguir iremos modificar o projeto atual para implementar melhorias.

Melhorando a solução

Embora a separação em classes tenha gerado um ponto positivo na solução, separando o que é regra de negócio de interface, a solução pode ser melhorada removendo do formulário a responsabilidade de saber como funciona cada implementação de envio de e-mail. Existe um princípio chamado SRP, que é o princípio da responsabilidade única, que recomenda que uma classe só deva ter um motivo para sofrer alguma alteração. Sendo assim, podemos resumir que uma classe qualquer em seu sistema deva ter uma tarefa única e específica. Isso, contudo, não quer dizer que se tenha apenas um único método, mas que todos os métodos criados trabalhem juntos em um único objetivo, atender a responsabilidade da classe. Outros autores também chamam este princípio de coesão.

Listagem 9. Evento OnClick do botão Enviar

```
01 procedure TfrmPrincipal.btnEnviarClick(Sender: TObject);
02 begin
03   case rgEmail.ItemIndex of
04     0 :
05     begin
06       _EmailIndy.RemetenteEmail := edtDe.Text;
07       _EmailIndy.DestinatarioEmail := edtPara.Text;
08       _EmailIndy.Assunto := edtAssunto.Text;
09       _EmailIndy.Host := edtSMTP.Text;
10       _EmailIndy.Usuario := edtUsuario.Text;
11       _EmailIndy.Senha := edtSenha.Text;
12       _EmailIndy.Porta := StrToInt(edtPorta.Text);
13       _EmailIndy.Mensagem := memTexto.Text;
14       _EmailIndy.Enviar();
15     end;
16     1 :
17     begin
18       _EmailSynapse.RemetenteEmail := edtDe.Text;
19       _EmailSynapse.DestinatarioEmail := edtPara.Text;
20       _EmailSynapse.Assunto := edtAssunto.Text;
21       _EmailSynapse.Host := edtSMTP.Text;
22       _EmailSynapse.Usuario := edtUsuario.Text;
23       _EmailSynapse.Senha := edtSenha.Text;
24       _EmailSynapse.Porta := StrToInt(edtPorta.Text);
25       _EmailSynapse.Mensagem := memTexto.Text;
26       _EmailSynapse.Enviar();
27     end;
28     2 :
29     begin
30       _EmailClever.RemetenteEmail := edtDe.Text;
31       _EmailClever.DestinatarioEmail := edtPara.Text;
32       _EmailClever.Assunto := edtAssunto.Text;
33       _EmailClever.Host := edtSMTP.Text;
34       _EmailClever.Usuario := edtUsuario.Text;
35       _EmailClever.Senha := edtSenha.Text;
36       _EmailClever.Porta := StrToInt(edtPorta.Text);
37       _EmailClever.Mensagem := memTexto.Text;
38       _EmailClever.Enviar();
39     end;
40   end;
41 end;
```

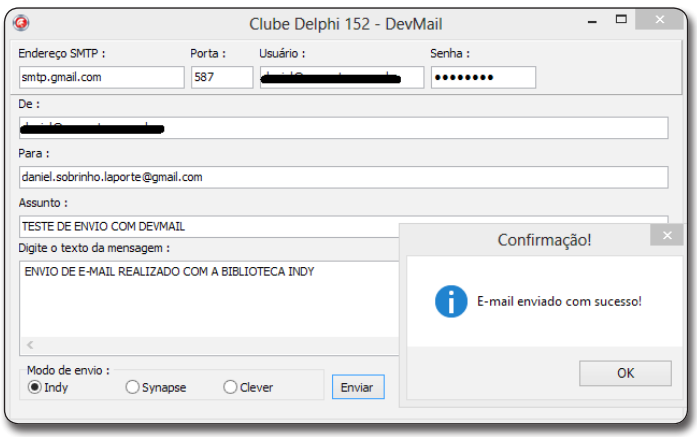

Figura 2. Envio de e-mail pelo DevMail

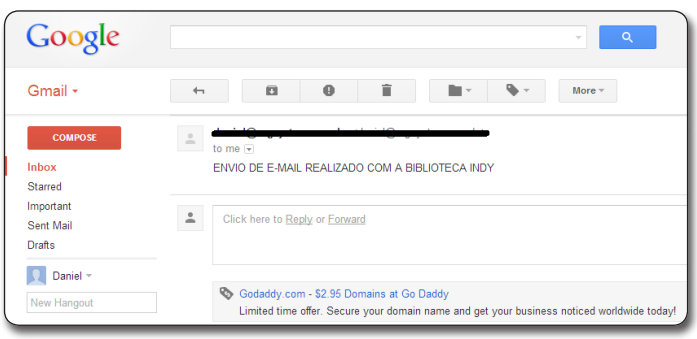

Figura 3. Recebendo o e-mail enviado

Para entendermos seu significado vamos imaginar um cenário que todos conhecem. Digamos que você possui uma classe que represente uma Nota Fiscal, a classe NotaFiscal. E você a codificou de tal forma que ela mesma, além de conter seus itens, seus cálculos, também sabe como se salvar e se recuperar do banco de dados e até mesmo sabe como exportar nos mais diversos formatos. Se pararmos para pensar, você acha que é responsabilidade da NotaFiscal saber tudo isso? Apesar da forma que foi construída, temos essa classe lidando com mais de uma situação de responsabilidades diferentes. Se amanhã a forma de exportação é alterada, é preciso corrigir a classe. Se também algo muda na regra de negócio referente a uma nota fiscal, essa classe também precisa ser corrigida, ou seja, temos aqui mais de um motivo para essa classe ser alterada, e isso não é bom. Não é bom porque cada responsabilidade é um arco de mudanças, que muito provavelmente estão entrelaçadas. Estamos falando aqui de responsabilidade, mas o que é isso para uma classe?

No contexto deste artigo, ela representa um motivo de mudança. Nem sempre é fácil identificar responsabilidades a mais em uma classe, isso porque estamos acostumados a ver como um todo. Em nosso exemplo a formulário principal, que é uma classe, está acumulando algo que não deveria, saber como funciona cada tipo de implementação de envio de e-mail. Isso é visto quando o formulário repetidamente acessa os membros de cada implementação. Essa responsabilidade precisa ser repassada para alguém, mas quem? É agora que os padrões de projeto vêm ao nosso socorro.

Ao analisar a solução, vemos que temos três algoritmos diferentes para uma mesma solução, contudo eles precisam ser encapsulados e se devem ser intercambiáveis, ou seja, a classe cliente que irá utilizá-los não deve perceber que em um dado momento a implementação com Indy está em funcionamento e em outro a versão Synapse está em execução. Temos um padrão de projeto que atua exatamente nessa situação.

O padrão Strategy

Dentro do catálogo definido por GoF (Ganf of Four), que são quatro autores que sintetizaram os padrões mais comuns em um livro, o strategy enquadra-se na seção comportamental.

Ou seja, ele vem permitir um comportamento dinâmico para uma aplicação, sem criar fortes dependências entre o código envolvido. Para implementação desse padrão algumas classes, possuindo papéis específicos precisam ser criadas. Temos como participantes:

• Strategy – Ou estratégia, declara uma interface comum para todos os algoritmos implementados. A classe com papel Context utiliza essa interface para chamar o algoritmo definido por uma classe com papel ConcreteStrategy;

• ConcreteStrategy – Estratégia concreta, implementa o algoritmo utilizando a interface de Strategy;

• Context – Contexto, é configurado com um objeto ConcreteStrategy, mantém uma referência ao objeto Strategy e pode definir uma interface que permite ao Strategy acessar seus dados.

Toda essa definição pode ser mais bem visualizada na **Figura 4** através de seu diagrama de classes.

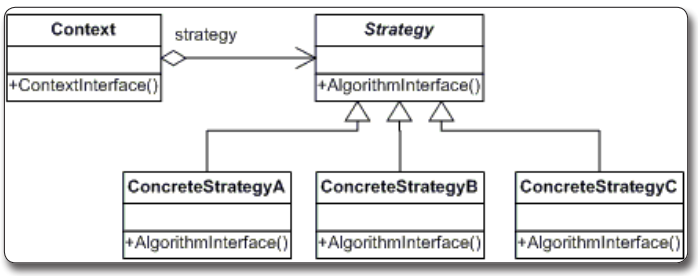

Figura 4. Diagrama de classes do padrão Strategy

Implementando o Strategy

A primeira modificação necessária no código existe é o encapsulamento das várias implementações de envio em uma interface comum. Dessa forma, modificamos a unit uInterfaceEmail adicionando a ela a interface da **Listagem 10**.

Listagem 10. Interface de estratégia

```
IEstrategiaEnvio = Interface
  ['{354DBF1C-7BBB-40C0-82B8-999BB4BB6B02}']
  procedure Enviar(email: IEmail; SSL : Boolean = True);
end;
```

A seguir, como já possuímos as implementações, vamos modificá-las para que, ao invés de implementarem a interface IEmail já existente, venham implementar IEstrategiaEnvio, como mostra a **Listagem 11**, que exibe o que foi modificado na classe TEmailIndy.

Como agora o método Enviar receber via parâmetro um objeto do tipo IEmail, podemos simplesmente passar seus dados para o objeto TEmailIndy. Cada implementação existente deve sofrer essa mesma alteração.

Ao observar as **Listagens 10 e 11** e comparar com a estrutura proposta pela **Figura 4** ficam claros os papéis das classes criadas até o momento. Podemos identificar que cada tipo de envio simboliza um ConcreteStrategy, que a nova interface IEstrategiaEnvio equivale a Strategy. Fica faltando apenas o Context, que já possuímos também, a interface IEmail. Nela temos a definição do que é comum em e-mail. Contudo, a definição de seu método Enviar precisa ser modificada para receber, além da indicação de um envio por SSL, o tipo de estratégia que será utilizado. Também foi criada uma classe que implementa IEmail, chamada EmailContexto como mostra a **Listagem 12**. Ela representa então o Context.

Listagem 11. Modificações em TEmailIndy

```
TEmailIndy = class(TInterfacedObject, IEstrategiaEnvio)
...
public
  procedure Enviar(email: IEmail; SSL : Boolean = True); overload;
...
end;

procedure TEmailIndy.Enviar(email: IEmail; SSL : Boolean = True);
begin
  RemetenteEmail := email.RemetenteEmail;
  DestinatarioEmail := email.DestinatarioEmail;
  Assunto := email.Assunto;
  Host := email.Host;
  Usuario := email.Usuario;
  Senha := email.Senha;
  Porta := email.Porta;
  Mensagem := email.Mensagem;
...
end;
```

Listagem 12. Modificação da interface IEmail

```
IEmail = Interface
...
  procedure Enviar(EstrategiaDeEnvio:IEstrategiaEnvio; SSL : Boolean = True);
end;

EmailContexto = class(TInterfacedObject, IEmail)
...
public
...
  procedure Enviar(EstrategiaDeEnvio:IEstrategiaEnvio; SSL : Boolean = True);
end;

procedure EmailContexto.Enviar(EstrategiaDeEnvio: IEstrategiaEnvio;
  SSL: Boolean);
begin
  FEstrategiaDeEnvio := EstrategiaDeEnvio;
  FEstrategiaDeEnvio.Enviar(self, SSL);
end;
```

Ao observar a implementação do método Enviar, vemos que quem executa o envio, é a estratégia passada como parâmetro, dessa forma o formulário principal também deverá sofrer alguns ajustes, como mostra a **Listagem 13**.

Nas linhas 3 e 4 temos as variáveis que representam a arquitetura criada, elas são do tipo interface para garantir a limpeza automática da memória quando não estiverem mais em uso. Quando o evento OnClick do botão Enviar for disparado, o objeto de contexto é criado, o EmailContexto (linha 6). A ele são passadas as informações digitadas no formulário. Então, verifica-se qual implementação de envio será utilizada (linha 16), e dependendo do escolhido uma

estratégia é criada (linhas 17, 18, 19) e então, essa estratégia é passada ao método Enviar (linha 22). Ao executar novamente o aplicativo vemos que tudo continua funcionando perfeitamente.

Listagem 13. Utilizando o Strategy

```
01 procedure TfrmPrincipal.Button2Click(Sender: TObject);
02 var
03   objEmail: IEmail;
04   objEstrategia: IEstrategiaEnvio;
05 begin
06   objEmail := EmailContexto.Create;
07   objEmail.RemetenteEmail := edtDe.Text;
08   objEmail.DestinatarioEmail := edtPara.Text;
09   objEmail.Assunto := edtAssunto.Text;
10   objEmail.Host := edtSMTP.Text;
11   objEmail.Usuario := edtUsuario.Text;
12   objEmail.Senha := edtSenha.Text;
13   objEmail.Porta := StrToInt(edtPorta.Text);
14   objEmail.Mensagem := memTexto.Text;
15
16   case rgEmail.ItemIndex of
17     0 : objEstrategia := TEmailIndy.Create(Self);
18     1 : objEstrategia := TEmailSynalist.Create(Self);
19     2 : objEstrategia := TEmailClever.Create(Self);
20   end;
21
22   objEmail.Enviar(objEstrategia);
23 end;
```

Conclusão

Ao criar um elemento que representa um contexto, IEmail, e torná-lo o ponto de conhecimento único do formulário ganhamos flexibilidade para criar novos tipos de implementação sem gerar grandes impactos no formulário e praticamente, sem impacto algum nessa classe de contexto. Contudo, a solução apresentada pode ser melhorada de tal forma que o formulário nem precise saber que existe uma classe TEmailIndy, TEmailSynalist e TEmailClever. Ela conheceria apenas IEstrategiaEnvio.

Para alcançar esse objetivo, bastaria abstrair a construção das estratégias utilizando outro padrão de projeto, por exemplo, o FactoryMethod. Fica o desafio ao leitor!

Autor

Daniel Sobrinho Laporte
daniel.laporte@gmail.com / Twitter: @daniel_laporte
Colunista e editor técnico da revista Clube Delphi tendo publicado mais de 40 artigos técnicos. Graduado em Sistemas de informação e pós-graduado em Administração de Bancos de Dados. Atualmente é analista de sistemas e desenvolvedor na empresa Orgsystem Software. Desenvolve em Delphi desde 2005 adotando como foco principal a arquitetura multicamadas com os mais variados bancos de dados.

Autor

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi e editor técnico .Net Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas da Siplan Control-M (www.siplancontrolm.com.br) unidade Jaboticabal, e professor da FATEC, campus Taquaritinga. Blog: http://quicoli.worpress.com

Links:

Synapse
http://synapse.ararat.cz

Clever Components
http://www.clevercomponents.com/

Indy
http://www.indyproject.org/index.en.aspx

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Aplicações de banco de dados - 1-tier, 2-tier e N-tier

Aprenda os conceitos envolvidos nestes tipos de aplicações

O desenvolvimento em camadas de uma aplicação de banco de dados representa essencialmente sua segmentação em partes lógicas e físicas. No Delphi, a visibilidade maior desta "divisão" se dá fundamentalmente na questão física, o que acaba por refletir diretamente sobre o número de processos em execução pelo contexto. De forma geral, neste cenário, cada camada representa um processo, caracterizando as aplicações conhecidas como 1-tier, 2-tier e N-tier. Para este último, o N faz menção a um número ilimitado de camadas, ficando a critério do desenvolvedor a sua definição. Todavia, especificamente no Delphi, o N comumente equivale a três camadas.

Dito isto, tendo como base as aplicações de banco de dados, é possível caracterizá-las da seguinte forma:

- 1 camada: aplicações que se utilizam de um banco de dados local;
- 2 camadas: aplicações que utilizam um servidor de banco de dados;
- 3 camadas: aplicações que utilizam um servidor de banco de dados e um servidor de aplicação.

Banco de dados

Em sua essência, um banco de dados pode ser definido como sendo uma coleção disciplinada de dados. No âmbito da programação, isso se reflete em uma coleção de registros (dados) que ficam armazenados na máquina, em arquivos computacionais. Toda a gerência de um banco de dados fica a cargo de um software específico, denominado Sistema de Gerenciamento de Banco de Dados, ou simplesmente SGBD.

A forma com que são estruturados os dados em um banco de dados incide sobre o seu "modelo". Dentre os existentes, o modelo tido como "relacional" é o que se destaca, sendo atualmente o mais usual e plenamente suportado pelo Delphi. Em sua abrangência, o modelo relacional estrutura as informações do banco em linhas e colunas, distribuídas em várias tabelas relacionadas.

Resumo DevMan

Aplicações de banco de dados - 1-tier, 2-tier e N-tier:

De seu início até os tempos atuais, o Delphi vem tendo uma evolução quase que natural em termos de tecnologia e recursos, sempre acompanhando as principais tendências de mercado. Só para citar, mais recentemente foram feitas melhorias em seu Core, o que inclui seus frameworks clássicos, como VCL e dbExpress, bem como a incorporação de recursos inéditos até então, vide FireMonkey e o suporte a Mac OS X e Win64. Embora tudo isso seja de fundamental importância no desenvolvimento de aplicações, alguns pontos tidos como básicos invariavelmente se mostram essenciais para o contexto da programação propriamente dita. Dentre estes, a arquitetura de software se destaca, uma vez que é a base para a construção das mais variadas aplicações. No Delphi, esta questão arquitetural é essencialmente refletida no desenvolvimento de aplicações 1, 2 ou N camadas, também conhecidas como 1-tier, 2-tier e N-tier. Partindo deste ponto, o artigo irá mostrar os conceitos técnicos que envolvem cada uma destas, a fim de se estabelecer um estudo comparatório entre elas, da teoria à prática.

Em que situação o tema é útil:

Historicamente falando, a grande notoriedade do Delphi se refere à sua excelência no que diz respeito ao desenvolvimento de aplicações de banco de dados para a plataforma Windows (Win32) sob a arquitetura Cliente/Servidor. Partindo disto, é importante ter em mente que a ferramenta provê ainda muitas outras possibilidades. Olhando especificamente para o aspecto arquitetural, além do modelo Cliente/Servidor, que é tido como um arquitetura de 2 camadas, se fazem presente ainda ao contexto Delphi as aplicações de 1 e N camadas. Devido a isso, torna-se de grande valia a qualquer desenvolvedor um estudo fundamental sobre cada uma destas alternativas, adquirindo o devido conhecimento sobre cada uma, sempre visando uma eventual utilização

Além do modelo, os bancos de dados também são segmentados pelo seu tipo. Por padrão, o Delphi provê suporte a dois tipos: local e remoto.

Banco de dados local

São bancos que residem localmente, seja em disco ou em rede. No popular, são também tratados como "bancos de dados baseados em arquivo". Alguns velhos conhecidos como o Paradox, dBase e Microsoft Access são alguns exemplos de banco de dados local.

Banco de dados remoto

São os bancos relacionados ao conceito de Servidor de Banco de Dados, uma vez que, comumente, residem em uma estação separada. Cada banco caracteriza um produto diferente e, consequentemente, por ser de fabricantes diferentes, variam em sua maneira de tratar e armazenar os dados. Em razão disso, todos os clientes trabalham sob uma interface lógica comum, que é a SQL **(BOX 1),** utilizada para se acessar/manipular bancos de dados relacionais. Nestas situações, o servidor de banco de dados passa a ser tratado, em algumas literaturas, como "servidor de banco de dados SQL". InterBase, Firebird, Oracle, Microsoft SQL Server e MySQL são alguns exemplos mais populares de banco de dados remoto.

BOX 1. SQL

Structured Query Language, ou simplesmente SQL, pode ser definida como sendo uma linguagem de consulta estruturada, originária dos anos 70. Desde então, tornou-se o padrão adotado para a realização de pesquisas para banco de dados relacional, em razão de sua simplicidade e facilidade de uso. Em outras palavras, é a linguagem mais utilizada para a criação, modificação e recuperação de dados em bancos de dados relacionais. Criada originalmente pela IBM, no decorrer do tempo e fruto de sua própria evolução, a Structured Query Language ganhou novas vertentes desenvolvidas por outros fabricantes, caracterizando assim vários dialetos. Em sua extensão, a SQL é segmentada em vários subconjuntos, de acordo com as operações efetuadas sobre um banco de dados, tais como:

- DML (Data Manipulation Language): Linguagem de manipulação de dados. Ex: Select, Insert e Update;
- DDL (Data Definition Language): Linguagem de definição de dados. Ex: Create, Alter e Drop;
- DCL (Data Control Language): Linguagem de controle de dados. Ex: Grant e Revoke;
- DTL (Data Transaction Language): Linguagem de transação de dados. Ex: Commit e Rollback.

Aplicações 1-tier

No âmbito do desenvolvimento de software, aplicações tidas como sendo de uma camada (1-tier/Single-tier) são aquelas que, essencialmente, alocam um único processo na máquina ou compartilham um único sistema de arquivos. Em suma, num mesmo elemento estarão sendo executadas as diversas partes que compõem um sistema. Dessa forma, na própria aplicação é que estarão implementadas todas as tratativas de dados, regras de negócio e apresentação (UI). No Delphi, aplicações de banco de dados baseadas em arquivos (de dados), também conhecidas como File-based data applications, podem ser consideradas as mais representativas deste cenário. Como característica principal, essas aplicações não fazem uso de um servidor de banco de dados para a manipulação de suas informações (dados). Ao invés disso, são usados arquivos de dados locais presentes no mesmo ambiente da aplicação.

Arquivos de dados

Olhando para uma abordagem mais clássica, é possível citar como exemplo as aplicações que faziam uso do já obsoleto Paradox como "banco de dados". Já para o atual momento, o foco das aplicações 1-tier se voltam à utilização do componente TClientDataSet vinculado diretamente a um arquivo dedicado de dados, o qual acaba por substituir o uso de um servidor de banco de dados. A **Figura 1** mostra um esboço do modelo citado. Nessa designação, um arquivo de dados fará o papel de uma tabela do banco de dados. Tais arquivos são tratados como arquivos MyBase (MyBase files) e são tidos como especiais, uma vez que atendem a uma padronização específica e exclusiva ao componente. Na prática, eles podem apresentar dois formatos distintos, mediante o suporte provido pelo ClientDataSet:

- XML: também tratados como MyBase XML Table, esses arquivos armazenam os dados num formato baseado nas especificações do XML, numa estruturação semelhante a uma tabela do banco de dados;
- Binário: também tratados como CDS (ClientDataSet) files, estes arquivos armazenam os dados num formato binário sendo, portanto, mais rápidos que o formato XML;

De forma nativa, o próprio Delphi já traz em sua instalação alguns arquivos deste tipo como exemplo. Por padrão, eles ficam localizados na parte de Documentos públicos do ambiente, no diretório "Samples\Data" da pasta relativa à versão instalada do Delphi. Citando o XE3 em um ambiente em inglês, o caminho seria algo como "C:\Users\Public\Documents\RAD Studio\10.0\Samples\Data".

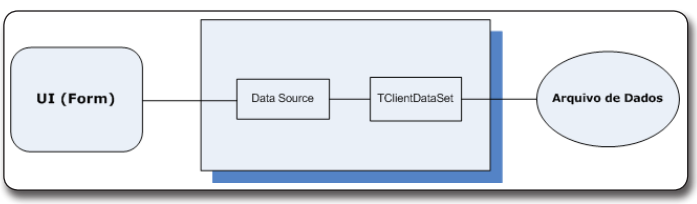

Figura 1. Modelo 1-tier

Aspectos práticos – trabalhando com arquivos de dados existentes

Em termos de código, em razão do uso do TClientDataSet, os métodos a serem utilizados para a manipulação dos dados nos arquivos são basicamente os mesmos do desenvolvimento tradicional com banco de dados, vide Append/Insert para inserção, Edit para edição, Delete para exclusão e etc. Além destes, este tipo de aplicação contempla ainda outros dois métodos complementares, denominados LoadFromFile e SaveToFile, designados para carregar/ler os dados para o ClientDataSet, bem como para persistir as alterações realizadas em memória para os arquivos em disco, respectivamente. Tais métodos são extremamente úteis nas situações em que o arquivo de dados já existe, e se deseja apenas carregá-lo ou salvá-lo a partir de um ClientDataSet.

Numa exemplificação simples, a **Listagem 1** mostra uma chamada a ambos. Na situação apresentada é utilizado o arquivo de dados "employee.cds" que acompanha o Delphi. O caminho completo deste arquivo é armazenado numa constante do sistema

denominada Arquivo. Por conseguinte, ela é utilizada como valor de parâmetro aos métodos citados. Partindo do princípio que o componente de dados esteja ligado a um TDBGrid, os dados carregados a partir do arquivo mediante a chamada ao método LoadFromFile seriam prontamente exibidos em tela, conforme mostra a **Figura 2**. Da mesma forma, o botão que faz a chamada ao método SaveToFile é quem irá efetivar no arquivo qualquer mudança realizada sobre os dados.

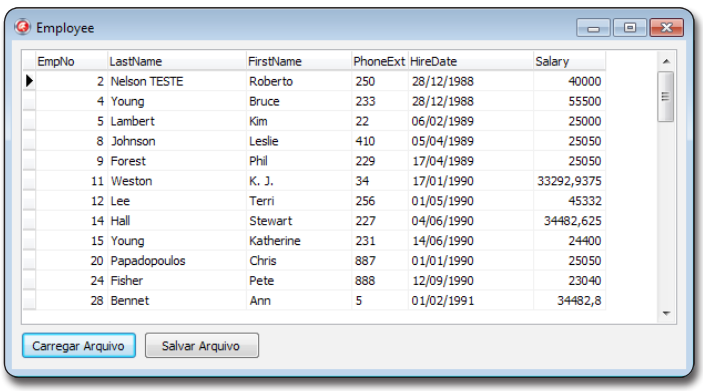

Figura 2. Dados do arquivo employee.cds sendo mostrados em tela

Listagem 1. ClientDataSet – LoadFromFile e SaveToFile

```
01 ...
02 const
03   Arquivo = 'C:\Users\Public\Documents\RAD
         Studio\10.0\Samples\Data\employee.cds';
04 ...

05 procedure TForm1.btnCarregarArquivoClick(Sender: TObject);
06 begin
07   ClientDataSet1.LoadFromFile(Arquivo);
08 end;

09 procedure TForm1.btnSalvarArquivoClick(Sender: TObject);
10 begin
11   ClientDataSet1.SaveToFile(Arquivo);
12 end;
```

Outra variação deste cenário envolve a indicação do nome do arquivo de dados diretamente no componente, substituindo a utilização do método LoadFromFile. Isso é feito a partir da propriedade FileName do componente, que provê uma caixa de diálogo de seleção, tal como mostra a **Figura 3**. A diferença nesta abordagem fica por conta do desenvolvimento em si. Uma vez que um arquivo de dados válido é definido nesta propriedade, os dados passam a ser automaticamente carregados a partir do arquivo, assim que o componente ClientDataSet é aberto. Para se salvar as alterações realizadas sobre os dados, a chamada ao método SaveToFile é substituída por uma simples chamada ao método Post, seguindo a linha do desenvolvimento tradicional de banco de dados, com o diferencial de que aqui os dados estão sendo salvos tanto em memória quanto em disco.

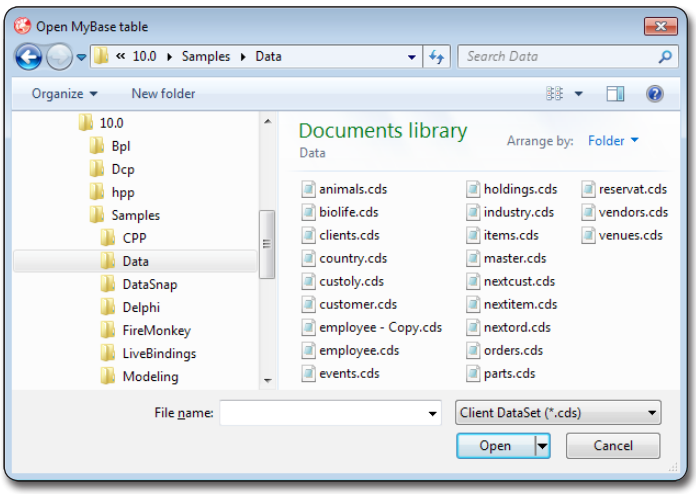

Figura 3. Propriedade FileName – selecionando arquivo de dados

Aspectos práticos – Definindo um novo arquivo de dados a partir de um TClientDataSet

No Delphi, ainda no contexto das aplicações de banco de dados baseadas em arquivo, é possível ainda fazer uso do próprio componente TClientDataSet para se estabelecer toda uma estrutura de dados e, posteriormente, salvá-la como arquivo. Obviamente, a grande vantagem aqui está relacionada ao total controle que o desenvolvedor tem sobre a estrutura que estará sendo criada, aliada à facilidade em se gerar um arquivo de dados a partir dela, o que descarta qualquer trabalho de produzi-lo manualmente.

Na prática, toda a estrutura de dados (metadados) é criada na lista de Fields do ClientDataSet, todavia, este processo se dá basicamente de duas formas: em tempo de Design, com auxílio do IDE, ou em tempo de execução, via código. A primeira forma pode ser considerada a mais trabalhosa, por envolver um processo repetitivo e manual e, ao mesmo tempo, pode ser considerada também a mais intuitiva, pelo fato de todo o processo ser feito de forma visual. Portanto, nesta abordagem, a definição dos "campos" que irão compor a estrutura ocorre no FieldsEditor do ClientDataSet, acessível pelo seu menu de contexto. Durante a adição de um novo Field, uma janela de diálogo é exibida, contemplando os detalhes do novo campo, conforme mostra a **Figura 4**. Uma vez definida a lista de campos, a efetivação da estrutura em memória é feita pela chamada ao método CreateDataSet. Novamente, isso pode ser feito ainda em tempo de Design, pelo mesmo menu de contexto. No momento seguinte, através dele também é possível se gerar o arquivo de dados, através das opções "Save to MyBase XML table" ou "Save to binary MyBase file". Uma vez gerado, este arquivo poderá então ser vinculado diretamente ao ClientDataSet, tal como já mostrado anteriormente.

Este mesmo processo também pode ser feito via código, definindo-se a estrutura de dados já em tempo de execução. A **Listagem 2** exibe um exemplo simples desta definição. De forma complementar, para se salvar a estrutura criada num arquivo em disco, entra em cena novamente o método SaveToFile do ClientDataSet. Pela sua assinatura, mostrada a seguir (**Listagem 3**),

fica fácil entender como utilizá-lo nestas situações. O detalhe fica por conta do último parâmetro (Format), que indica o formato em que o arquivo será salvo, se XML ou binário.

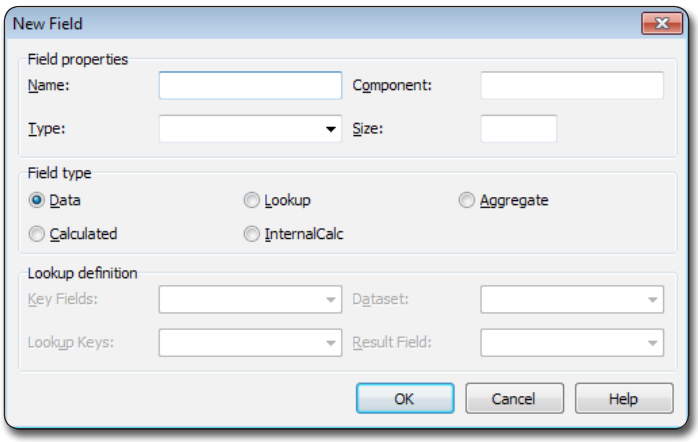

Figura 4. Adicionando um novo Field no ClientDataSet

Listagem 2. Criando, via código, a estrutura de dados no ClientDataSet

```
01 ...
02 ClientDataSet1.FieldDefs.Clear;
03 ClientDataSet1.FieldDefs.Add('CODIGO', ftInteger, 0, False);
04 ClientDataSet1.FieldDefs.Add('REVISTA', ftString, 80, False);
05 ClientDataSet1.FieldDefs.Add('CAPA', ftString, 80, False);
06 ClientDataSet1.FieldDefs.Add('COMPLEXIDADE', ftInteger, 0, False);
07 ClientDataSet1.CreateDataSet;
08 ...
```

Listagem 3. Definição do método SaveToFile do ClientDataSet

```
01 procedure SaveToFile(const FileName: string = ''; Format:
   TDataPacketFormat = dfBinary);
```

Aplicações 2-tier

No Delphi, aplicações 2-tier são expressas nas tradicionais aplicações Client/Server (Cliente/Servidor). O ponto chave dessas aplicações é a utilização de um servidor de banco de dados para a parte Server da arquitetura, responsável pela tratativa dos dados, em complemento à parte Client, que determina a própria aplicação. Este cenário acaba por definir dois segmentos distintos, caracterizando as duas camadas. De forma simples, além do processo alocado pela aplicação, o servidor de banco de dados constitui outro processo independente, o que viabiliza sua execução tanto localmente quanto remotamente (em outra máquina).

O Delphi provê diversas opções, em termos de tecnologia, para a construção deste tipo de aplicação. Em seu estágio atual, duas tecnologias se destacam: dbExpress e FireDAC. Isso se deve ao fato de ambas proverem um acesso rápido, por meio de drivers específicos, a diversos servidores de banco de dados, tais como: InterBase, Firebird, Oracle, Microsoft SQL Server, MySQL, entre outros.

dbExpress e ClientDataSet

O dbExpress acompanha o Delphi desde a versão 6 do IDE e atualmente ocupa o status de principal tecnologia nativa para acesso a dados. Este destaque pode ser confirmado, por exemplo, na presença da janela Data Explorer (**Figura 5**) no IDE. Através dela o desenvolvedor é capaz de definir e manipular conexões dbExpress aos diversos servidores de banco de dados suportados. Já olhando pelo lado técnico, outra característica fundamental do dbExpress é o seu trabalho com DataSets unidirecionais, que proporciona uma maior rapidez no acesso aos dados do banco. Em contrapartida, esta mesma característica produz certas limitações ao mecanismo. Só para citar, conforme a própria nomeação sugere, DataSets unidirecionais só permitem a navegação dos dados em sentido único, do primeiro ao último registro, de forma sequencial. Essa limitação pode ser observada facilmente ao se tentar, por exemplo, ligar um DataSet do dbExpress a um controle TDBGrid. Neste momento é levantada uma exceção, ainda em tempo de Design, conforme mostra a **Figura 6**. Além disso, DataSets unidirecionais não proveem suporte à edição de dados, em decorrência da ausência de recursos de buffer de dados em memória.

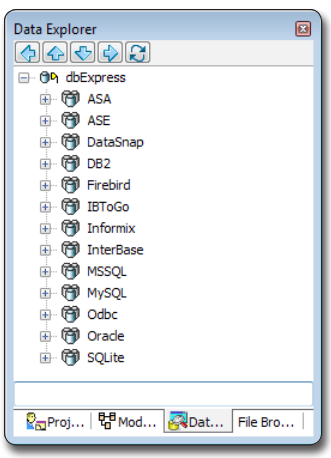

Figura 5. Data Explorer

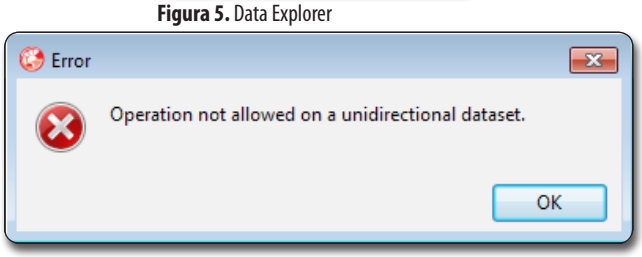

Figura 6. Exceção lançada ao se tentar vincular um DataSet unidirecional a um DBGrid

Mediante essas limitações, em aplicações de banco de dados, a utilização do dbExpress é complementada pelo uso de componentes da guia DataAccess da Tool Palette, mais especificamente TDataSetProvider e TClientDataSet. Neste contexto, o Provider exerce o papel de elemento intermediário entre um DataSet do dbExpress (TSQLDataSet, TSQLQuery, etc.) e o ClientDataSet, ficando responsável tanto pela requisição de pacotes de dados da fonte (DataSet) quanto pela resolução da atualização dos mesmos

no banco. Já o ClientDataSet é quem irá gerenciar os dados em memória, eliminando assim as restrições impostas pelo dbExpress citadas anteriormente. Com isso, torna-se possível a navegação fluída (bidirecional) sobre os dados, bem como a parte de persistência. Expressando visualmente tudo que foi dito, a **Figura 7** mostra um cenário mínimo presente no desenvolvimento de uma aplicação de banco de dados no Delphi que faça uso do dbExpress para acesso a dados.

Em complemento ao que foi exposto anteriormente, aqui o TSQLConnection é o componente responsável por toda a parte de conexão com o banco de dados. A partir disso, os DataSets (neste caso TSQLDataSet) se ligam a ele a fim de recuperar conjunto de dados provenientes dos objetos do banco, tais como tabelas e Stored Procedures. A seguir na **Listagem 4** é mostrado o código do arquivo .dfm dos componentes, que expressa a ligação existente entre cada elemento.

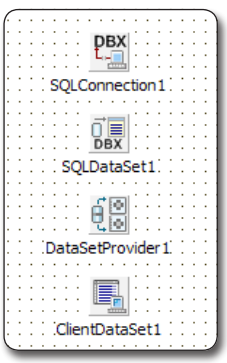

Figura 7. dbExpress – Componentes de acesso a dados

Listagem 4. Visualização do arquivo .dfm dos componentes da **Figura 7**

```
01 object SQLDataSet1: TSQLDataSet
02   SQLConnection = SQLConnection1
03   ...
04 end

05 object DataSetProvider1: TDataSetProvider
06   DataSet = SQLDataSet1
07   ...
08 end

09 object ClientDataSet1: TClientDataSet
10   ProviderName = 'DataSetProvider1'
11   ...
12 end
```

A **Figura 8** mostra um esboço do modelo 2-tier mencionado.

FireDAC

O FireDAC é uma poderosa biblioteca de componentes de acesso a dados, oriundo da empresa DA-Soft Technologies, onde se apresentava sob o nome de AnyDAC. Recentemente adquirido pela Embarcadero, foi rebatizado, surgindo como uma das últimas grandes novidades incorporadas ao Delphi ainda na versão XE3. Em termos funcionais, numa comparação quase que inevitável, é nítida sua semelhança com o dbExpress, principalmente em razão de sua capacidade em prover acesso nativo a diversos servidores de banco de dados. Na mesma medida, há também algumas diferenças entre ambos. Uma das principais diz respeito aos DataSets que, ao contrário do dbExpress, no FireDAC são bidirecionais e com suporte a atualização direta de dados no banco. Sendo assim, não se torna essencial o uso complementar de componentes como TDataSetProvider e TClientDataSet.

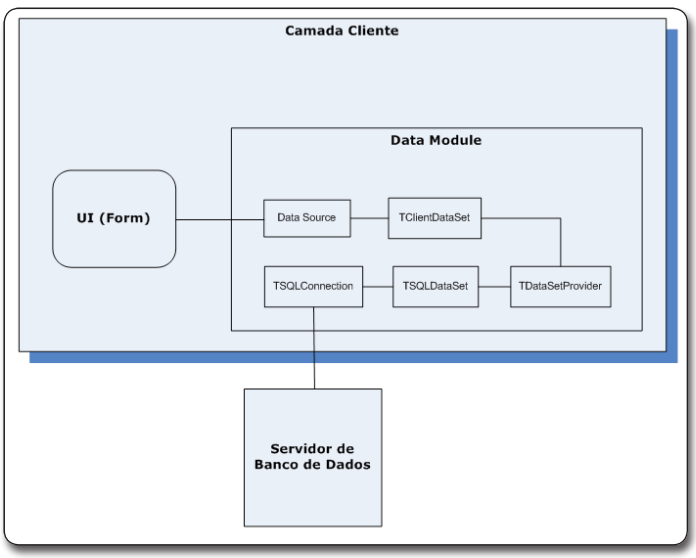

Figura 8. Modelo 2-tier – dbExpress

No lado prático, tendo em vista a conexão com uma base de dados Firebird e, por conseguinte, o acesso a uma de suas tabelas, a forma mais básica de uso do FireDAC em aplicações de banco de dados 2-tier no Delphi se dá pelo envolvimento dos componentes mostrados na **Figura 9**:

- TADConnection: componente responsável pela parte de conexão com o banco de dados;
- TADQuery: componente DataSet que irá recuperar um conjunto de dados do banco a partir de uma instrução SQL;
- TADPhysIBDriverLink: componente que fica a cargo da conexão com a biblioteca cliente do banco de dados utilizado. No FireDAC, sua utilização substitui a necessidade de se distribuir DLLs de drivers do framework, tal como acontece com o dbExpress;
- TADGUIxWaitCursor: componente que controla o cursor de espera (wait cursor) relacionado às ações efetuadas na aplicação;

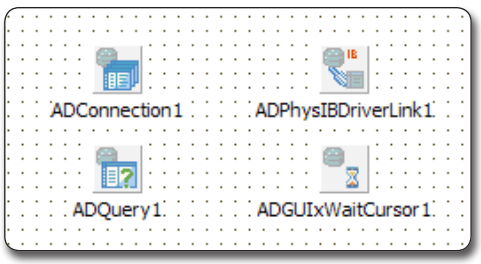

Figura 9. FireDAC – Componentes de acesso a dados

Aplicações 3-tier

No contexto Delphi, os termos 3-tier, N-tier ou Multi-tier são utilizados para se designar um tipo de aplicação que segue um modelo popularmente conhecido como multicamadas. Se comparado ao modelo tradicional de 2 camadas (Cliente/Servidor), as quais representam a Aplicação Cliente e o Servidor de Banco de Dados, seu diferencial fica por conta da presença de uma camada extra, representada por um elemento denominado Servidor de Aplicação. Na prática, trata-se de uma aplicação que irá residir no servidor, estando acessível às eventuais aplicações clientes. Por este fato, é nela que deverão estar concentradas e implementadas toda a parte de acesso a dados e regras de negócio do âmbito do software. Por conseguinte, o cliente passa a ser um mero meio de interação com o usuário final, contendo basicamente validações de tela (UI). Por este motivo, neste modelo as aplicações cliente são tidas como "thin-clients" (clientes magros).

Pelo fato de uma aplicação multicamadas inserir um novo elemento ao cenário, o que acaba por inevitavelmente, aumentar a complexidade de desenvolvimento, torna-se de suma importância ter o conhecimento sobre as vantagens proporcionadas por um Servidor de Aplicação. A seguir, são listadas algumas delas:

• O custo com licenças de banco de dados pagos é reduzido drasticamente, isto porque o modelo multicamadas faz com que a aplicação cliente não mais se comunique diretamente com o servidor de banco de dados, e sim com o Servidor de Aplicação. Num cenário otimista, o Servidor de Aplicação faz uso de uma única conexão, que será "compartilhada" por todos os Clientes;

• O Servidor de Aplicação encapsula toda a parte de acesso a dados, sendo assim, uma eventual mudança na tecnologia utilizada (de dbExpress para FireDAC, por exemplo) em nada irá afetar as aplicações cliente;

• No Servidor de Aplicação é onde estarão concentradas todas as regras de negócio envolvidas. Em razão disso, ele é o único elemento que irá sofrer alterações e deverá ser atualizado, quando há mudanças nas regras. As aplicações cliente nada sofrem;

• O Servidor de Aplicação reside em uma máquina servidora pelo fato de exigir um hardware mais robusto, capaz de suportar toda execução da parte de acesso a dados e regras de negócio do contexto do software. Isso livra as estações clientes do processamento mais custoso, o que acaba por exigir menos recursos de hardware, consequentemente, reduzindo custos com equipamentos.

DataSnap

No estágio atual do Delphi, a tecnologia responsável por prover a construção de aplicações multicamadas é o DataSnap, tanto que a própria Embarcadero o denomina como sendo a "solução de aplicações de banco de dados multicamadas". Historicamente falando, o DataSnap nos remete às primeiras versões do IDE, tendo surgido ainda no Delphi 3, sob o nome de MIDAS (Multi-tier Distributed Application Services Suite). Vivendo de uma constante evolução, foi no Delphi 2009 que a tecnologia sofreu uma grande reestruturação, tendo como base o dbExpress e extinguindo a dependência com a tecnologia COM. Devido a essas mudanças em sua arquitetura, o DataSnap 2009 passou inclusive a ser tratado como "Novo DataSnap".

Olhando pelo lado prático, o uso do DataSnap no desenvolvimento de aplicações multicamadas está muito relacionado a utilização de outros dois elementos: dbExpress e TClientDataSet. O primeiro, para a parte de acesso, tanto de dados quanto ao Servidor de Aplicação, e o segundo para as tratativas de dados no lado Cliente. Um esboço do cenário multicamadas utilizando DataSnap pode ser visto na **Figura 10**.

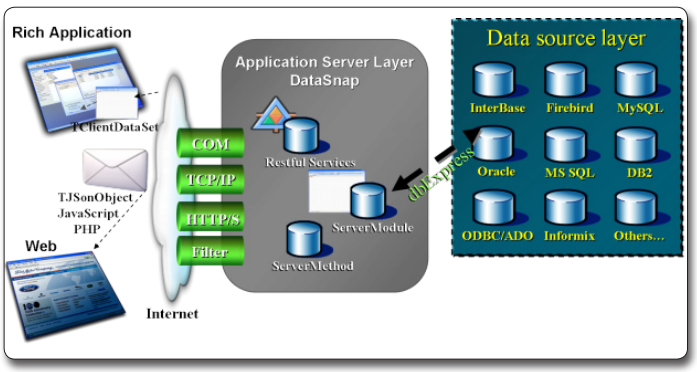

Figura 10. Modelo multicamadas com DataSnap (Fonte: site da embarcadero)

Servidor de Aplicação DataSnap

Um Servidor de Aplicação DataSnap nada mais é do que uma aplicação Delphi tradicional, contendo alguns elementos-chave adequados ao modelo multicamadas. Sendo assim, ela pode ser de três tipos: VCL Forms Application, Console Application ou Service Application. A escolha pelo tipo ideal de aplicação varia de acordo com a necessidade e objetivo do contexto ao qual está se trabalhando. Numa exemplificação simples, uma aplicação VCL Forms seria a ideal para desenvolvimento e testes, enquanto que uma aplicação do tipo Service é mais recomendada para ambientes de produção, uma vez que não necessita de intervenção humana para ser iniciada.

Em termos de elementos-chave, no lado servidor é que estão inseridos os principais itens para o ideal funcionamento da mecânica do DataSnap. Em se tratando de componentes, podemos destacar alguns deles, todos os presentes do grupo DataSnap Server da ToolPalette do IDE (**Figura 11**):

• DSServer: principal componente de configuração do servidor DataSnap, responsável, dentre outras coisas, pela gerência de classes do servidor;

• DSServerClass: componente usado para expor os recursos públicos de uma classe do lado servidor para clientes remotos. Aliado a isso, é responsável também por determinar o ciclo de vida de cada instância desta classe;

• TDSTCPServerTransport: componente que habilita o transporte baseado em soquete (TCP) ao servidor;

• TDSHTTPService: componente que habilita a invocação de métodos do servidor por meio de uso do protocolo REST (**BOX 2**), a partir de um cliente baseado em browser.

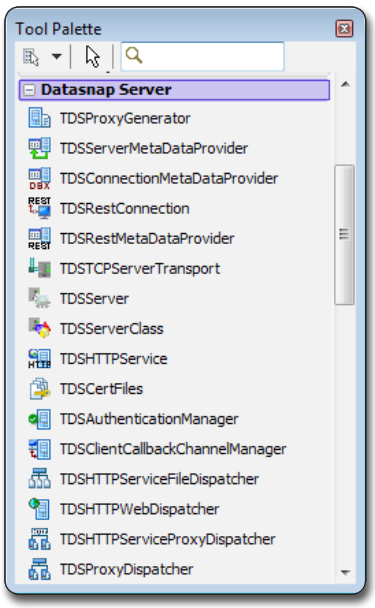

Figura 11. Componentes DataSnap Server

BOX 2. REST

REST (Representational State Transfer) pode ser brevemente definida como uma arquitetura Cliente/Servidor voltada para Web, por meio da World Wide Web. Em razão disso, seu protocolo subentende que o servidor esteja habilitado a trabalhar com quatro tipos de solicitações padrão: GET, POST, PUT e DELETE, equivalentes às operações de dados Retrieve, Update, Insert e Delete, respectivamente. Cada método disponibilizado pelo servidor poderá então ser invocado por meio do uso destas operações, por meio de um mecanismo específico, que se baseia na URI (Uniform Resource Indeitifier) e nos detalhes (nome e parâmetros) do próprio método.

Elucidando um cenário mínimo, tal como o mostrado no exemplo Client/Server, com o Servidor de Aplicação sendo então voltado a aplicações clientes Desktop, a **Figura 12** exibe os componentes DataSnap a serem usados no lado Server, nesta situação.

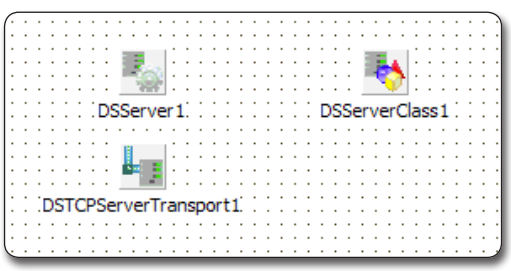

Figura 12. Componentes DataSnap no Servidor de Aplicação

De forma complementar, a seguir, na **Listagem 5** é mostrado por meio do código .dfm como se dá a ligação entre eles. Conforme pode ser visto, o componente TDSServer é requerido a todos os outros.

Acesso a dados

Conforme dito, toda a parte de acesso a dados também se concentra no Servidor de Aplicação. Em termos de componentes, pouca coisa muda em relação ao que é utilizado em aplicações Client/Server. Tomando como base o cenário apresentado pela **Figura 7**, a diferença inicial se dá pela retirada do componente TClientDataSet que, eventualmente, deverá ser deslocado para a parte cliente da arquitetura. Aliado a isso, é necessário se estabelecer uma forma de se expor os DataSets, via TDataSetProvider, ao ClientDataSet (agora presente na aplicação cliente). Uma solução simples para esta última situação é o uso de um novo container, particular ao DataSnap, em substituição ao atual TDataModule. Trata-se do TDSServerModule, que também descende de TDataModule, mas que contempla todas as características adicionais necessárias para se expor DataSets para as aplicações clientes. A **Figura 13** mostra como ficaria o TDataModule utilizado no exemplo 2-tier, agora num cenário 3-tier, utilizando TServerModule.

Listagem 5. Ligação dos componentes responsáveis pela elaboração do servidor de aplicação

```
01 object DSServer1: TDSServer
02 ...
03 end

04 object DSServerClass1: TDSServerClass
05 ...
06   Server = DSServer1
07 end

08 object DSTCPServerTransport1: TDSTCPServerTransport
09 ...
10   Server = DSServer1
11 end
```

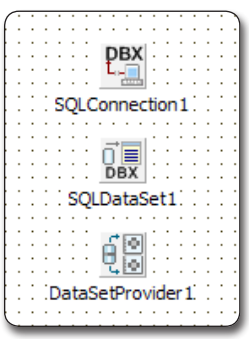

Figura 13. Componentes de acesso a dados no Server Module

A forma com que o componente TDSServerClass expõe uma classe do servidor aos clientes remotos se dá pelo seu evento OnGetClass, apontado por um parâmetro denominado PersistentClass. Sendo assim, ainda conforme o exemplo, a exposição da classe do TServerModule (nomeada aqui como TSMServer) seria algo como descrito na **Listagem 6**.

Aplicação Cliente

A partir de um Servidor de Aplicação, é possível se atender aos mais variados tipos de clientes, incluindo aplicações escritas em outras linguagens, como Java e C#, e até mesmo de outras

plataformas, como aplicações para Web e Mobile. Em se tratando do exemplo abordado, a aplicação Cliente poderia ser expressa por meio de um software Desktop Delphi do tipo VCL Forms. Para este cenário, conforme já citado, o dbExpress se faz presente, provendo o elemento (TSQLConnection) que fará a "conexão" com o Servidor de Aplicação, cuja configuração gira em torno das seguintes propriedades:

• DriverName: nome do driver a ser utilizado pela conexão. O dbExpress possui um driver específico para DataSnap, o qual é exigido neste tipo de aplicação;
• Params->HostName: nome do Host do servidor onde reside o Servidor de Aplicação;
• Params->Port: número da porta relativa ao que foi configurado no Servidor de Aplicação;
• CommunicationProtocol: tipo de protocolo utilizado na comunicação com o Servidor;

A seguir, na **Listagem 7** é mostrado um exemplo de configuração a ser utilizada neste exemplo:

Listagem 6. Exposição da classe através da variável PersistentClass

```
01 procedure TDMServer.DSServerClass1GetClass(DSServerClass: TDSServerClass;
   var PersistentClass: TPersistentClass);
02 begin
03   PersistentClass := TSMServer;
04 end;
```

Listagem 7. Exemplo de configuração do componente SQLConnection e driver DataSnap

```
01 object SQLConnection1: TSQLConnection
02    DriverName = 'DataSnap'
03    Params.Strings = (
04      …
05    'HostName=localhost'
06    'Port=211'
07    'CommunicationProtocol=tcp/ip')
08    …
09 end
```

A fim de se obter os pacotes de dados dos DataSets do servidor por meio dos Providers, no lado cliente entra em cena um novo componente específico, presente no grupo de componentes Datasnap Client, denominado TDSProviderConnection. Conforme seu nome sugere, ele é quem irá prover a conectividade necessária para, por exemplo, alimentar o TClientDataSet da parte cliente com um TDataSetProvider remoto. Para este exemplo, teríamos então a configuração mostrada a seguir, onde a propriedade ServerClassName do ProviderConnetion recebe o nome da classe exposta pelo servidor (TSMServer), enquanto que no ClientDataSet, em sua propriedade RemoteServer é apontado o próprio componente TDSProviderConnection, o que habilita o acesso ao TDataSetProvider (DataSetProvider1) do servidor. Observe a configuração conforme **Listagem 8**.

Feito isso, o ClientDataSet, presente no lado cliente, já está apto a receber/atualizar os pacotes de dados provindos do SQLDataSet, localizado no lado servidor através do DataSetProvider exposto.

Listagem 8. Configuração da conexão do DSProviderConnection e ClientDataSet (aplicação Cliente)

```
01 object DSProviderConnection1: TDSProviderConnection
02    ServerClassName = 'TSMServer'
03    SQLConnection = SQLConnection1
04 ...
05 end

06 object ClientDataSet1: TClientDataSet
07 ...
08   ProviderName = 'DataSetProvider1'
09   RemoteServer = DSProviderConnection1
10 ...
11 end
```

Conclusão

O presente artigo teve como principal objetivo expor de forma sucinta e prática, todo o conceito que envolve as três principais vertentes arquiteturais utilizadas no desenvolvimento de aplicações de banco de dados. Conforme visto, no Delphi, a construção de aplicações de 1, 2 ou N camadas se assemelham em termos de simplicidade, ao mesmo tempo em que se distinguem em funcionalidade. A partir do que foi mostrado, fica nítida a importância do componente TClientDataSet em meio ao desenvolvimento destas aplicações, justificando sua fama de "principal componente do Delphi", conforme apontado por muitos.

Além disso, este artigo também visou trazer para primeiro plano alguns dos aspectos mais tradicionais que circundam o desenvolvimento de aplicações no Delphi, ofuscadas em muitos momentos pela gama de novidades que surgem a todo instante. O constante estudo sobre lógica de programação, linguagem e arquitetura são itens essenciais e facilitadores a todo desenvolvimento de software.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links e Referências:

Delphi XE3 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE3 – Página oficial do produto
http://www.embarcadero.com/products/delphi

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

Clube Delphi

A Revista Clube Delphi oferece aos seus assinantes
uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS

- Curso de Multicamadas com Delphi e DataSnap
- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

http://www.devmedia.com.br

DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486